



JAKARTA EE

Jakarta XML Web Services

Jakarta XML Web Services Team, <https://projects.eclipse.org/projects/ee4j.jaxws>

3.0-RC3, May 29, 2020

Table of Contents

Eclipse Foundation Specification License	1
Disclaimers	2
1. Scope	3
Jakarta XML-Based Web Services, Version 2.3	4
2. Introduction	5
2.1. Goals	5
2.2. Non-Goals	7
2.3. Requirements	8
2.3.1. Relationship To JAXB	8
2.3.2. Standardized WSDL Mapping	8
2.3.3. Customizable WSDL Mapping	8
2.3.4. Standardized Protocol Bindings	9
2.3.5. Standardized Transport Bindings	9
2.3.6. Standardized Handler Framework	9
2.3.7. Versioning and Evolution	9
2.3.8. Standardized Synchronous and Asynchronous Invocation	9
2.3.9. Session Management	10
2.4. Use Cases	10
2.4.1. Handler Framework	10
2.4.1.1. Reliable Messaging Support	10
2.4.1.2. Message Logging	10
2.4.1.3. WS-I Conformance Checking	10
2.5. Conventions	10
2.6. Expert Group Members	12
2.7. Acknowledgements	13
3. WSDL 1.1 to Java Mapping	14
3.1. Definitions	14
3.1.1. Extensibility	15
3.2. Port Type	15
3.3. Operation	16
3.3.1. Message and Part	17
3.3.1.1. Non-wrapper Style	19
3.3.1.2. Wrapper Style	19
3.3.1.3. Example	21
3.3.2. Parameter Order and Return Type	23
3.3.3. Holder Class	24

3.3.4. Asynchrony	24
3.3.4.1. Standard Asynchronous Interfaces	24
3.3.4.2. Operation	24
3.3.4.3. Message and Part	25
3.3.4.4. Response Bean	25
3.3.4.5. Faults	26
3.3.4.6. Mapping Examples	27
3.3.4.7. Usage Examples	28
3.4. Types	28
3.4.1. W3CEndpointReference	29
3.5. Fault	29
3.5.1. Example	30
3.6. Binding	31
3.6.1.	32
3.6.2. SOAP Binding	32
3.6.2.1. Header Binding Extension	32
3.6.3. MIME Binding	33
3.6.3.1. mime:content	33
3.7. Service and Port	36
3.7.1. Example	38
3.8. XML Names	40
3.8.1. Name Collisions	40
4. Java to WSDL 1.1 Mapping	42
4.1. Java Names	42
4.1.1. Name Collisions	42
4.2. Package	42
4.3. Class	43
4.4. Interface	44
4.4.1. Inheritance	44
4.5. Method	44
4.5.1. One Way Operations	48
4.5.2. wsam:Action Computation Algorithm	49
4.6. Method Parameters and Return Type	50
4.6.1. Parameter and Return Type Classification	50
4.6.2. Use of JAXB	51
4.6.2.1. Document Wrapped	52
4.6.2.2. Document Bare	54
4.6.2.3. RPC	55

4.7. Service Specific Exception	56
4.8. Bindings	58
4.8.1. Interface	59
4.8.2. Method and Parameters	59
4.9. Generics	60
4.10. SOAP HTTP Binding	63
4.10.1. Interface	64
4.10.2. Method and Parameters	64
4.11. Service and Ports	67
5. Client APIs	69
5.1. javax.xml.ws.Service	69
5.1.1. Service Usage	69
5.1.1.1. Dynamic case	69
5.1.1.2. Static case	70
5.1.2. Provider and Service Delegate	71
5.1.3. Handler Resolver	72
5.1.4. Executor	73
5.2. javax.xml.ws.BindingProvider	73
5.2.1. Configuration	74
5.2.1.1. Standard Properties	75
5.2.1.2. Additional Properties	76
5.2.2. Asynchronous Operations	77
5.2.3. Proxies	77
5.2.3.1. Example	78
5.2.4. Exceptions	79
5.3. javax.xml.ws.Dispatch	79
5.3.1. Configuration	81
5.3.2. Operation Invocation	81
5.3.3. Asynchronous Response	82
5.3.4. Using JAXB	83
5.3.5. Examples	83
5.3.5.1. Synchronous, Payload-Oriented	83
5.3.5.2. Synchronous, Message-Oriented	83
5.3.5.3. Synchronous, Payload-Oriented With JAXB Objects	83
5.3.5.4. Asynchronous, Polling, Message-Oriented	84
5.3.5.5. Asynchronous, Callback, Payload-Oriented	84
5.4. Catalog Facility	84
5.5. javax.xml.ws.EndpointReference	85

6. Service APIs	87
6.1. javax.xml.ws.Provider	87
6.1.1. Invocation	88
6.1.1.1. Exceptions	88
6.1.2. Configuration	88
6.1.3. Examples	88
6.1.3.1. Simple echo service, reply message is the same as the input message	88
6.1.3.2. Simple static reply, reply message contains a fixed acknowledgment element	89
6.1.3.3. Using JAXB to read the input message and set the reply	89
6.2. javax.xml.ws.Endpoint	89
6.2.1. Endpoint Usage	90
6.2.2. Publishing	91
6.2.2.1. Example	92
6.2.3. Publishing Permission	93
6.2.4. Endpoint Metadata	93
6.2.5. Determining the Contract for an Endpoint	93
6.2.5.1. SEI-based Endpoints	94
6.2.5.2. Provider-based Endpoints	94
6.2.5.3. Use of @WebService(wsdlLocation) and Metadata	95
6.2.5.4. Application-specified Service	97
6.2.5.5. Application-specified PortType	97
6.2.5.6. Application-specified Schema or No Metadata	97
6.2.6. Endpoint Properties	98
6.2.7. Executor	98
6.2.8. javax.xml.ws.EndpointReference	98
6.3. javax.xml.ws.WebServiceContext	99
6.3.1. MessageContext	100
6.4. javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder	101
7. Core APIs	102
7.1. javax.xml.ws.Binding	102
7.2. javax.xml.ws.spi.Provider	102
7.2.1. Configuration	103
7.2.2. Creating Endpoint Objects	103
7.2.3. Creating ServiceDelegate Objects	104
7.2.4. EndpointReferences	104
7.2.5. Getting Port Objects	105
7.3. javax.xml.ws.spi.ServiceDelegate	105
7.4. Exceptions	105

7.4.1. Protocol Specific Exception Handling	106
7.4.1.1. Client Side Example	106
7.4.1.2. Server Side Example	106
7.4.1.3. One-way Operations	106
7.5. javax.xml.ws.WebServiceFeature	107
7.5.1. javax.xml.ws.soap.AddressingFeature	107
7.5.1.1. javax.xml.ws.EndpointReference	108
7.5.1.2. javax.xml.ws.W3CEndpointReference	108
7.5.1.3. javax.xml.ws.soap.MTOMFeature	108
7.5.1.4. javax.xml.ws.RespectBindingFeature	109
7.6. javax.xml.ws.spi.http (HTTP SPI)	109
8. Annotations	112
8.1. javax.xml.ws.ServiceMode	113
8.2. javax.xml.ws.WebFault	113
8.3. javax.xml.ws.RequestWrapper	113
8.4. javax.xml.ws.ResponseWrapper	114
8.5. javax.xml.ws.WebServiceClient	114
8.6. javax.xml.ws.WebEndpoint	115
8.6.1. Example	115
8.7. javax.xml.ws.WebServiceProvider	117
8.8. javax.xml.ws.BindingType	117
8.9. javax.xml.ws.WebServiceRef	117
8.9.1. Example	119
8.10. javax.xml.ws.WebServiceRefs	121
8.10.1. Example	121
8.11. Annotations Defined by JSR-181	122
8.11.1. javax.jws.WebService	122
8.11.2. javax.jws.WebMethod	122
8.11.3. javax.jws.OneWay	122
8.11.4. javax.jws.WebParam	123
8.11.5. javax.jws.WebResult	123
8.11.6. javax.jws.SOAPBinding	123
8.11.7. javax.jws.HandlerChain	123
8.12. javax.xml.ws.Action	124
8.13. javax.xml.ws.FaultAction	124
8.14. javax.xml.ws.spi.WebServiceFeatureAnnotation	125
8.14.1. javax.xml.ws.soap.Addressing	125
8.14.2. javax.xml.ws.soap.MTOM	126

8.14.3. javax.xml.ws.RespectBinding	126
9. Customizations	128
9.1. Binding Language	128
9.2. Binding Declaration Container	128
9.3. Embedded Binding Declarations	129
9.3.1. Example	129
9.4. External Binding File	129
9.4.1. Example	131
9.5. Using JAXB Binding Declarations	132
9.6. Scoping of Bindings	133
9.7. Standard Binding Declarations	133
9.7.1. Definitions	133
9.7.1.1. Semantics	134
9.7.2. PortType	134
9.7.2.1. Semantics	135
9.7.3. PortType Operation	135
9.7.3.1. Semantics	136
9.7.4. PortType Fault Message	137
9.7.4.1. Semantics	137
9.7.5. Binding	137
9.7.5.1. Semantics	137
9.7.6. Binding Operation	137
9.7.6.1. Semantics	138
9.7.7. Service	138
9.7.7.1. Semantics	138
9.7.8. Port	139
9.7.8.1. Semantics	139
10. Handler Framework	140
10.1. Architecture	140
10.1.1. Types of Handler	140
10.1.2. Binding Responsibilities	141
10.1.2.1. Handler and Message Context Management	141
10.1.2.2. Message Dispatch	142
10.1.2.3. Exception Handling	142
10.2. Configuration	143
10.2.1. Programmatic Configuration	143
10.2.1.1. javax.xml.ws.handler.HandlerResolver	143
10.2.1.2. Handler Ordering	143

10.2.1.3. javax.jws.HandlerChain annotation	144
10.2.1.4. javax.xml.ws.Binding	145
10.2.2. Deployment Model	145
10.3. Processing Model	145
10.3.1. Handler Lifecycle	145
10.3.2. Handler Execution	146
10.3.2.1. handleMessage	147
10.3.2.2. handleFault	148
10.3.2.3. close	149
10.3.3. Handler Implementation Considerations	149
10.4. Message Context	149
10.4.1. javax.xml.ws.handler.MessageContext	150
10.4.1.1. Standard Message Context Properties	150
10.4.2. javax.xml.ws.handler.LogicalMessageContext	154
10.4.3. Relationship to Application Contexts	154
11. SOAP Binding	156
11.1. Configuration	156
11.1.1. Programmatic Configuration	156
11.1.1.1. SOAP Roles	156
11.1.1.2. SOAP Handlers	157
11.1.1.3. SOAP Headers	158
11.1.2. Deployment Model	158
11.2. Processing Model	158
11.2.1. SOAP mustUnderstand Processing	158
11.2.2. Exception Handling	159
11.2.2.1. Handler Exceptions	159
11.2.2.2. Service Endpoint Exceptions	160
11.2.2.3. Mapping Exceptions to SOAP Faults	160
11.3. SOAP Message Context	161
11.4. SOAP Transport and Transfer Bindings	161
11.4.1. HTTP	161
11.4.1.1. MTOM	161
11.4.1.2. One-way Operations	162
11.4.1.3. Security	162
11.4.1.4. Session Management	163
11.4.1.5. Addressing	163
12. HTTP Binding	165
12.1. Configuration	165

12.1.1. Programmatic Configuration	165
12.1.1.1. HTTP Handlers	165
12.1.2. Deployment Model	166
12.2. Processing Model	166
12.2.1. Exception Handling	166
12.2.1.1. Handler Exceptions	166
12.2.1.2. Service Endpoint Exceptions	167
12.2.1.3. Mapping Exceptions to a HTTP Status Code	167
12.3. HTTP Support	167
12.3.1. One-way Operations	167
12.3.2. Security	167
12.3.3. Session Management	168
Appendix A: Conformance Requirements	169
Appendix B: Change Log	170
2.3 Changes	170
2.2 Changes	170
2.1 Changes	171
Changes since Proposed Final Draft	172
Changes since Public Draft	173
Changes Since Early Draft 3	175
Changes Since Early Draft 2	176
Changes Since Early Draft 1	176
Bibliography	179

Specification: Jakarta XML Web Services

Version: 3.0-RC3

Status: DRAFT

Release: May 29, 2020

Copyright (c) 2019 Eclipse Foundation.

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [\$date-of-document] Eclipse Foundation, Inc. [\[url to this license\]](#)"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) 2018 Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Chapter 1. Scope

Jakarta XML Web Services defines a means for implementing XML-Based Web Services based on Jakarta SOAP with Attachments and Jakarta Web Services Metadata.

Jakarta XML-Based Web Services, Version 2.3

Comments to <javaee-spec@javaee.groups.io> v2.3, July 11, 2017 :decription: Jakarta XML-Based Web Services, Version 2.3 :sectnums: :sectnumlevels: 5 :xrefstyle: full

Editors:

Jitendra Kotamraju

Lukas Jungmann

Comments to: javaee-spec@javaee.groups.io

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065 USA

Copyright (c) 2017, 2019 Oracle and/or its affiliates. All rights reserved.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Chapter 2. Introduction

XML[1] is a platform-independent means of representing structured information. XML Web Services use XML as the basis for communication between Web-based services and clients of those services and inherit XML's platform independence. SOAP[2][3][4] describes one such XML based message format and ``defines, using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols."

WSDL[5] is ``an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information." WSDL can be considered the de-facto service description language for XML Web Services.

JAX-RPC 1.0[6] defined APIs and conventions for supporting RPC oriented XML Web Services in the Java platform. JAX-RPC 1.1[7] added support for the WS-I Basic Profile 1.0[8] to improve interoperability between JAX-RPC implementations and with services implemented using other technologies.

JAX-WS 2.0 (this specification) is a follow-on to JAX-RPC 1.1, extending it as described in the following sections.

2.1. Goals

Since the release of JAX-RPC 1.0[6], new specifications and new versions of the standards it depends on have been released. JAX-WS 2.0 relates to these specifications and standards as follows:

JAXB

Due primarily to scheduling concerns, JAX-RPC 1.0 defined its own data binding facilities. With the release of JAXB 1.0 [9] there is no reason to maintain two separate sets of XML mapping rules in the Java platform. JAX-WS 2.0 will delegate data binding-related tasks to the JAXB 2.0[10] specification that is being developed in parallel with JAX-WS 2.0. JAXB 2.0[10] will add support for Java to XML mapping, additional support for less used XML schema constructs, and provide bidirectional customization of Java \rightarrow XML data binding. JAX-WS 2.0 will allow full use of JAXB provided facilities including binding customization and optional schema validation.

- JAX-WS 2.1 requires JAXB 2.1[11].
- JAX-WS 2.2 requires JAXB 2.2[12].
- JAX-WS 2.3 requires JAXB 2.3[13].

SOAP 1.2

Whilst SOAP 1.1 is still widely deployed, it's expected that services will migrate to SOAP 1.2[3][4] now that it is a W3C Recommendation. JAX-WS 2.0 will add support for SOAP 1.2 whilst requiring continued support for SOAP 1.1.

WSDL 2.0

The W3C is expected to progress WSDL 2.0[\[14\]](#) to Recommendation during the lifetime of this JSR. JAX-WS 2.0 will add support for WSDL 2.0 whilst requiring continued support for WSDL 1.1. + The expert group for the JSR decided against this goal for this release . We will look at adding support in a future revision of the JAX-WS specification.

WS-I Basic Profile 1.1

JAX-RPC 1.1 added support for WS-I Basic Profile 1.0. WS-I Basic Profile 1.1 is expected to supersede 1.0 during the lifetime of this JSR and JAX-WS 2.0 will add support for the additional clarifications it provides.

A Metadata Facility for the Java Programming Language (JSR 175)

JAX-WS 2.0 will define the use of Java annotations[\[15\]](#) to simplify the most common development scenarios for both clients and servers.

Web Services Metadata for the Java Platform (JSR 181)

JAX-WS 2.0 will align with and complement the annotations defined by JSR 181[\[16\]](#).

Implementing Enterprise Web Services (JSR 109)

The JSR 109[\[17\]](#) defined `jaxrpc-mapping-info` deployment descriptor provides deployment time Java \rightarrow WSDL mapping functionality. In conjunction with JSR 181[\[16\]](#), JAX-WS 2.0 will complement this mapping functionality with development time Java annotations that control Java \rightarrow WSDL mapping.

Web Services Security (JSR 183)

JAX-WS 2.0 will align with and complement the security APIs defined by JSR 183[\[18\]](#).

JAX-WS 2.0 will improve support for document/message centric usage:

Asynchrony

JAX-WS 2.0 will add support for client side asynchronous operations.

Non-HTTP Transports

JAX-WS 2.0 will improve the separation between the XML message format and the underlying transport mechanism to simplify use of JAX-WS with non-HTTP transports.

Message Access

JAX-WS 2.0 will simplify client and service access to the messages underlying an exchange.

Session Management

JAX-RPC 1.1 session management capabilities are tied to HTTP. JAX-WS 2.0 will add support for message based session management.

JAX-WS 2.0 will also address issues that have arisen with experience of implementing and using JAX-RPC 1.0:

Inclusion in Java SE

JAX-WS 2.0 will prepare JAX-WS for inclusion in a future version of Java SE. Application portability is a key requirement and JAX-WS 2.0 will define mechanisms to produce fully portable clients.

Handlers

JAX-WS 2.0 will simplify the development of handlers and will provide a mechanism to allow handlers to collaborate with service clients and service endpoint implementations.

Versioning and Evolution of Web Services

JAX-WS 2.0 will describe techniques and mechanisms to ease the burden on developers when creating new versions of existing services.

2.2. Non-Goals

The following are non-goals:

Backwards Compatibility of Binary Artifacts

Binary compatibility between JAX-RPC 1.x and JAX-WS 2.0 implementation runtimes.

Pluggable data binding

JAX-WS 2.0 will defer data binding to JAXB[10]; it is not a goal to provide a plug-in API to allow other types of data binding technologies to be used in place of JAXB. However, JAX-WS 2.0 will maintain the capability to selectively disable data binding to provide an XML based fragment suitable for use as input to alternative data binding technologies.

SOAP Encoding Support

Use of the SOAP encoding is essentially deprecated in the web services community, e.g., the WS-I Basic Profile[8] excludes SOAP encoding. Instead, literal usage is preferred, either in the RPC or document style. + SOAP 1.1 encoding is supported in JAX-RPC 1.0 and 1.1 but its support in JAX-WS 2.0 runs counter to the goal of delegation of data binding to JAXB. Therefore JAX-WS 2.0 will make support for SOAP 1.1 encoding optional and defer description of it to JAX-RPC 1.1. + Support for the SOAP 1.2 Encoding[4] is optional in SOAP 1.2 and JAX-WS 2.0 will not add support for SOAP 1.2 encoding.

Backwards Compatibility of Generated Artifacts

JAX-RPC 1.0 and JAXB 1.0 bind XML to Java in different ways. Generating source code that works with unmodified JAX-RPC 1.x client source code is not a goal.

Support for Java versions prior to Java SE 5.0

JAX-WS 2.0 relies on many of the Java language features added in Java SE 5.0. It is not a goal to support JAX-WS 2.0 on Java versions prior to Java SE 5.0.

Service Registration and Discovery

It is not a goal of JAX-WS 2.0 to describe registration and discovery of services via UDDI or ebXML

RR. This capability is provided independently by JAXR[\[19\]](#).

2.3. Requirements

2.3.1. Relationship To JAXB

JAX-WS describes the WSDL \rightarrow Java mapping, but data binding is delegated to JAXB [\[10\]](#). The specification must clearly designate where JAXB rules apply to the WSDL \rightarrow Java mapping without reproducing those rules and must describe how JAXB capabilities (e.g., the JAXB binding language) are incorporated into JAX-WS. JAX-WS is required to be able to influence the JAXB binding, e.g., to avoid name collisions and to be able to control schema validation on serialization and deserialization.

2.3.2. Standardized WSDL Mapping

WSDL is the de-facto service description language for XML Web Services. The specification must specify a standard WSDL \rightarrow Java mapping. The following versions of WSDL must be supported:

- WSDL 1.1[\[5\]](#) as clarified by the WS-I Basic Profile(Ballinger, Ehnebuske, Gudgin, et al. 2004; Ballinger, Ehnebuske, Ferris, et al. 2004)

The standardized WSDL mapping will describe the default WSDL \rightarrow Java mapping. The default mapping may be overridden using customizations as described below.

2.3.3. Customizable WSDL Mapping

The specification must provide a standard way to customize the WSDL \rightarrow Java mapping. The following customization methods will be specified:

Java Annotations

In conjunction with JAXB[\[10\]](#) and JSR 181[\[16\]](#), the specification will define a set of standard annotations that may be used in Java source files to specify the mapping from Java artifacts to their associated WSDL components. The annotations will support mapping to WSDL 1.1.

WSDL Annotations

In conjunction with JAXB[\[10\]](#) and JSR 181[\[16\]](#), the specification will define a set of standard annotations that may be used either within WSDL documents or as in an external form to specify the mapping from WSDL components to their associated Java artifacts. The annotations will support mapping from WSDL 1.1.

The specification must describe the precedence rules governing combinations of the customization methods.

2.3.4. Standardized Protocol Bindings

The specification must describe standard bindings to the following protocols:

- SOAP 1.1[2] as clarified by the WS-I Basic Profile[8][20]
- SOAP 1.2[3][4]

The specification must not prevent non-standard bindings to other protocols.

2.3.5. Standardized Transport Bindings

The specification must describe standard bindings to the following protocols:

- HTTP/1.1[21].

The specification must not prevent non-standard bindings to other transports.

2.3.6. Standardized Handler Framework

The specification must include a standardized handler framework that describes:

Data binding for handlers

The framework will offer data binding facilities to handlers and will support handlers that are decoupled from the SAAJ API.

Handler Context

The framework will describe a mechanism for communicating properties between handlers and the associated service clients and service endpoint implementations.

Unified Response and Fault Handling

The `handleResponse` and `handleFault` methods will be unified and the declarative model for handlers will be improved.

2.3.7. Versioning and Evolution

The specification must describe techniques and mechanisms to support versioning of service endpoint interfaces. The facilities must allow new versions of an interface to be deployed whilst maintaining compatibility for existing clients.

2.3.8. Standardized Synchronous and Asynchronous Invocation

There must be a detailed description of the generated method signatures to support both asynchronous and synchronous method invocation in stubs generated by JAX-WS. Both forms of invocation will support a user configurable timeout period.

2.3.9. Session Management

The specification must describe a standard session management mechanism including:

Session APIs

Definition of a session interface and methods to obtain the session interface and initiate sessions for handlers and service endpoint implementations.

HTTP based sessions

The session management mechanism must support HTTP cookies and URL rewriting.

SOAP based sessions

The session management mechanism must support SOAP based session information.

2.4. Use Cases

2.4.1. Handler Framework

2.4.1.1. Reliable Messaging Support

A developer wishes to add support for a reliable messaging SOAP feature to an existing service endpoint. The support takes the form of a JAX-WS handler.

2.4.1.2. Message Logging

A developer wishes to log incoming and outgoing messages for later analysis, e.g., checking messages using the WS-I testing tools.

2.4.1.3. WS-I Conformance Checking

A developer wishes to check incoming and outgoing messages for conformance to one or more WS-I profiles at runtime.

2.5. Conventions

The keywords `MUST`, `MUST NOT`, `REQUIRED`, `SHALL`, `SHALL NOT`, `SHOULD`, `SHOULD NOT`, `RECOMMENDED`, `MAY`, and `OPTIONAL` in this document are to be interpreted as described in RFC 2119[\[22\]](#).

For convenience, conformance requirements are called out from the main text as follows:

Conformance (Example): Implementations **MUST** do something.

A list of all such conformance requirements can be found in appendix [Conformance Requirements](#).

Java code and XML fragments are formatted as shown in figure [\[1.1\]](#):

Figure 1.1: Example Java Code

```
package com.example.hello;

public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

Non-normative notes are formatted as shown below.

Note: *This is a note.*

This specification uses a number of namespace prefixes throughout; they are listed in Table [Table 1.1]. Note that the choice of any namespace prefix is arbitrary and not semantically significant (see XML Infoset[23]).

Table 1.1: Prefixes and Namespaces used in this specification.

Prefix	Namespace	Notes
env	http://www.w3.org/2003/05/soap-envelope	A normative XML Schema [24][25] document for the http://www.w3.org/2003/05/soap-envelope namespace can be found at http://www.w3.org/2003/05/soap-envelope .
xsd	http://www.w3.org/2001/XMLSchema	The namespace of the XML schema[24][25] specification
wsdl	http://schemas.xmlsoap.org/wsdl/	The namespace of the WSDL schema[5]
soap	http://schemas.xmlsoap.org/wsdl/soap/	The namespace of the WSDL SOAP binding schema[24][25]
jaxb	http://java.sun.com/xml/ns/jaxb	The namespace of the JAXB [9] specification
jaxws	http://java.sun.com/xml/ns/jaxws	The namespace of the JAX-WS specification
wsa	http://www.w3.org/2005/08/addressing	The namespace of the WS-Addressing 1.0[26] schema
wsam	http://www.w3.org/2007/05/addressing/metadata	The namespace of the WS-Addressing 1.0 - Metadata[27] schema

wsp	http://www.w3.org/ns/ws-policy	The namespace of the Web Services Policy 1.5 - Framework[28] schema
-----	---	---

Namespace names of the general form `http://example.org/...` and `http://example.com/...` represent application or context-dependent URIs (see RFC 2396[21]).

All parts of this specification are normative, with the exception of examples, notes and sections explicitly marked as `Non-Normative`.

2.6. Expert Group Members

The following people have contributed to this specification:

Chavdar Baikov (SAP AG)
 Russell Butek (IBM)
 Manoj Cheenath (BEA Systems)
 Shih-Chang Chen (Oracle)
 Claus Nyhus Christensen (Trifork)
 Ugo Corda (SeeBeyond Technology Corp)
 Glen Daniels (Sonic Software)
 Alan Davies (SeeBeyond Technology Corp)
 Thomas Diesler (JBoss, Inc.)
 Jim Frost (Art Technology Group Inc)
 Alastair Harwood (Cap Gemini)
 Marc Hadley (Sun Microsystems, Inc.)
 Kevin R. Jones (Developmentor)
 Lukas Jungmann (Oracle)
 Anish Karmarkar (Oracle)
 Toshiyuki Kimura (NTT Data Corp)
 Jim Knutson (IBM)
 Doug Kohlert (Sun Microsystems, Inc)
 Daniel Kulp (IONA Technologies PLC)
 Sunil Kunisetty (Oracle)
 Changshin Lee (Tmax Soft, Inc)
 Carlo Marcoli (Cap Gemini)
 Srividya Natarajan (Nokia Corporation)
 Sanjay Patil (SAP AG)
 Greg Pavlik (Oracle)
 Bjarne Rasmussen (Novell, Inc)
 Sebastien Sahuc (Intalio, Inc.)
 Rahul Sharma (Motorola)
 Rajiv Shivane (Pramati Technologies)
 Richard Sitze (IBM)
 Dennis M. Sosnoski (Sosnoski Software)

Christopher St. John (WebMethods Corporation)

Mark Stewart (ATG)

Neal Yin (BEA Systems)

Brian Zotter (BEA Systems)

Nicholas L Gallardo (IBM)

Alessio Soldano (Red Hat)

2.7. Acknowledgements

Robert Bissett, Arun Gupta, Graham Hamilton, Mark Hapner, Jitendra Kotamraju, Vivek Pandey, Santiago Pericas-Geertsens, Eduardo Pelegri-Llopart, Rama Pulavarthi, Paul Sandoz, Bill Shannon, and Kathy Walsh (all from Sun Microsystems) have provided invaluable technical input to the JAX-WS 2.0 specification.

Roberto Chinnici, Marc Hadley, Kohsuke Kawaguchi, and Bill Shannon (all from Sun Microsystems) have provided invaluable technical input to the JAX-WS 2.2 specification. I would like to thank Rama Pulavarthi for his contributions to the 2.2 reference implementation and to the specification. JAX-WS TCK team (Arthur Frechette, Alan Frechette) and SQE team (Jonathan Benoit) assisted the conformance testing of the 2.2 specification.

Chapter 3. WSDL 1.1 to Java Mapping

This chapter describes the mapping from WSDL 1.1 to Java. This mapping is used when generating web service interfaces for clients and endpoints from a WSDL 1.1 description.

Conformance (WSDL 1.1 support): Implementations MUST support mapping WSDL 1.1 to Java.

The following sections describe the default mapping from each WSDL 1.1 construct to the equivalent Java construct. In WSDL 1.1, the separation between the abstract port type definition and the binding to a protocol is not complete. Bindings impact the mapping between WSDL elements used in the abstract port type definition and Java method parameters. Section [wsdl11bindingmap] describes binding dependent mappings.

An application MAY customize the mapping using embedded binding declarations (see section [Embedded Binding Declarations](#)) or an external binding file (see section [External Binding File](#)).

Conformance (Customization required): Implementations MUST support customization of the WSDL 1.1 to Java mapping using the JAX-WS binding language defined in chapter [customize].

In order to enable annotations to be used at runtime for method dispatching and marshalling, this specification requires generated Java classes and interfaces to be annotated with the Web service annotations described in section [mdjsr181]. The annotations present on a generated class MUST faithfully reflect the information in the WSDL document(s) that were given as input to the mapping process, as well as the customizations embedded in them and those specified via any external binding files.

Conformance (Annotations on generated classes): The values of all the properties of all the generated annotations MUST be consistent with the information in the source WSDL document and the applicable external binding files.

3.1. Definitions

A WSDL document has a root `wsdl:definitions` element. A `wsdl:definitions` element and its associated `targetNamespace` attribute is mapped to a Java package. JAXB[10] (see appendix D) defines a standard mapping from a namespace URI to a Java package name. By default, this algorithm is used to map the value of a `wsdl:definitions` element's `targetNamespace` attribute to a Java package name.

Conformance (Definitions mapping): In the absence of customizations, the Java package name is mapped from the value of a `wsdl:definitions` element's `targetNamespace` attribute using the algorithm defined by JAXB[10].

An application MAY customize this mapping using the `jaxws:package` binding declaration defined in section [bindingdecldefinitions].

No specific authoring style is required for the input WSDL document; implementations should support WSDL that uses the WSDL and XML Schema import directives.

Conformance (WSDL and XML Schema import directives): Implementations MUST support the WS-I Basic Profile 1.1[20] defined mechanisms (See R2001, R2002, and R2003) for use of WSDL and XML Schema import directives.

3.1.1. Extensibility

WSDL 1.1 allows extension elements and attributes to be added to many of its constructs. JAX-WS specifies the mapping to Java of the extensibility elements and attributes defined for the SOAP and MIME bindings. JAX-WS does not address mapping of any other extensibility elements or attributes and does not provide a standard extensibility framework though which such support could be added in a standard way. Future versions of JAX-WS might add additional support for standard extensions as these become available.

Conformance (Optional WSDL extensions): An implementation MAY support mapping of additional WSDL extensibility elements and attributes not described in JAX-WS.

Note that such support may limit interoperability and application portability.

3.2. Port Type

A WSDL port type is a named set of abstract operation definitions. A `wsdl:portType` element is mapped to a Java interface in the package mapped from the `wsdl:definitions` element (see section [wsdl11defmap] for a description of `wsdl:definitions` mapping). A Java interface mapped from a `wsdl:portType` is called a *Service Endpoint Interface* or SEI for short.

Conformance (SEI naming): In the absence of customizations, the name of an SEI MUST be the value of the `name` attribute of the corresponding `wsdl:portType` element mapped according to the rules described in section [xmlnames].

An application MAY customize this mapping using the `jaxws:class` binding declaration defined in section [bindingdeclporttype].

Conformance (`javax.jws.WebService` required): A mapped SEI MUST be annotated with a `javax.jws.WebService` annotation.

A WSDL may define additional types via type substitution that are not referenced by a service directly but may still need to be marshalled by JAX-WS. The `javax.xml.bind.XmlSeeAlso` annotation from JAXB is used on the generated SEI to specify any additional types from the WSDL.

Conformance (`javax.xml.bind.XmlSeeAlso` required): An SEI generated from a WSDL that defines types not directly referenced by the `Port` MUST contain the `javax.xml.bind.XmlSeeAlso` annotation with all of the additional types referenced either directly or indirectly.


```
// Types generated when importing WSDL
package example;
public class A { ... }

package example1;
public class B extends A { ... }

package example2;
public class C extends A { ... }

// Directly annotated SEI with classes B and C
@WebService
@XmlSeeAlso({B.class, C.class})
public interface MyService {
    public A echo(A a);
}

// Indirectly annotated SEI using generated JAXB ObjectFactories
@XmlSeeAlso({example1.ObjectFactory.class, example2.ObjectFactory.class})
public interface MyService {
    public A echo(A a);
}
```

Figure 2.1: Directly and indirectly @XmlSeeAlso annotated SEI

Figure [2.1] shows how an SEI can be annotated with `javax.xml.bind.XmlSeeAlso`. This figure shows some of the types that may have been created while importing a WSDL and the different approaches to annotating the SEI.

An SEI contains Java methods mapped from the `wsdl:operation` child elements of the corresponding `wsdl:portType`, see section [Operation](#) for further details on `wsdl:operation` mapping. WSDL 1.1 does not support port type inheritance so each generated SEI will contain methods for all operations in the corresponding port type.

3.3. Operation

Each `wsdl:operation` in a `wsdl:portType` is mapped to a Java method in the corresponding Java service endpoint interface.

Conformance (Method naming): In the absence of customizations, the name of a mapped Java method MUST be the value of the `name` attribute of the `wsdl:operation` element mapped according to the rules described in section [xmlnames].

An application MAY customize this mapping using the `jaxws:method` binding declaration defined in section [PortType Operation](#).

Conformance (`javax.jws.WebMethod` required): A mapped Java method MUST be annotated with a `javax.jws.WebMethod` annotation. The annotation MAY be omitted if all its properties would have the default values.

The WS-I Basic Profile[20] R2304 requires that operations within a `wsdl:portType` have unique values for their `name` attribute so mapping of WS-I compliant WSDL descriptions will not generate Java interfaces with overloaded methods. However, for backwards compatibility, JAX-WS supports operation name overloading provided the overloading does not cause conflicts (as specified in the Java Language Specification[29]) in the mapped Java service endpoint interface declaration.

Conformance (Transmission primitive support): An implementation MUST support mapping of operations that use the `one-way` and `request-response` transmission primitives.

Conformance (Using `javax.jws.OneWay`): A Java method mapped from a one-way operation MUST be annotated with a `javax.jws.OneWay` annotation.

Mapping of `notification` and `solicit-response` operations is out of scope.

3.3.1. Message and Part

Each `wsdl:operation` refers to one or more `wsdl:message` elements via child `wsdl:input`, `wsdl:output`, and `wsdl:fault` elements that describe the input, output, and fault messages for the operation respectively. Each operation can specify one input message, zero or one output message, and zero or more fault messages.

Fault messages are mapped to application specific exceptions (see section [faulttoexceptmap]). The contents of input and output messages are mapped to Java method parameters using two different styles: non-wrapper style and wrapper style. The two mapping styles are described in the following subsections. Note that the binding of a port type can affect the mapping of that port type to Java, see section [wsdl11bindingmap] for details.

Conformance (Using `javax.jws.SOAPBinding`): An SEI mapped from a port type that is bound using the WSDL SOAP binding MUST be annotated with a `javax.jws.SOAPBinding` annotation describing the choice of style, encoding and parameter style. The annotation MAY be omitted if all its properties would have the default values (i.e. document/literal/wrapped).

Conformance (Using `javax.jws.WebParam`): Generated Java method parameters MUST be annotated with a `javax.jws.WebParam` annotation. If the style is `rpc` or if the style is `Document` and the parameter style is `BARE` then the `partName` element of `javax.jws.WebParam` MUST refer to the `wsdl:part` name of the parameter.

Conformance (Using `javax.jws.WebResult`): Generated Java methods MUST be annotated with a `javax.jws.WebResult` annotation. If the style is `rpc` or if the style is `Document` and the parameter style is `BARE` then the `partName` element of `javax.jws.WebResult` MUST refer to the `wsdl:part` name of the parameter. The annotation MAY be omitted if all its properties would have the default values.

WSDL description may have `wsam:Action` attribute on `wsdl:input`, `wsdl:output`, `wsdl:fault` elements in a

wsdl:operation. This **wsam:Action** attribute is used to explicitly define the value of the WS-Addressing Action header and this needs to be mapped on to the corresponding Java method.

Conformance (Generating @Action): Generated Java Methods MUST be annotated with **@Action** and **@FaultAction** annotations for the corresponding **wsdl:input**, **wsdl:output** and **wsdl:fault** messages that contain **wsam:Action** attributes

Conformance (Generating @Action input): If a **wsdl:input** element contains a **wsam:Action** attribute, the value of the attribute MUST be set to the **input** element of **@Action**

Conformance (Generating @Action output): If a **wsdl:output** element contains a **wsam:Action** attribute, the value of the attribute MUST be set to the **output** element of **@Action**

Conformance (Generating @Action fault): If a **wsdl:fault** element contains a **wsam:Action** attribute, the value of the attribute MUST be set to the value element of **@FaultAction**. The **className** element of **@FaultAction** MUST be the exception class name associated with the **wsdl:fault**

Figure 2.2: shows an example of the **wsam:Action** mapping described above.

```
<operation name="getPrice">
  <input message="tns:getPrice" />
  <output message="tns:getPriceResponse" wsam:Action="outAction"/>
  <fault name="InvalidTickerException"
    message="tns:InvalidTickerException" wsam:Action="faultAction"/>
</operation>

// the mapped java method
@Action(output="outAction",
    fault={ @FaultAction(className=InvalidTickerException.class,
        value="faultAction") }
)
float getPrice(String ticker) throws InvalidTickerException;
```

Figure 2.2: Mapping of wsam:Action metadata

When generating an SEI from WSDL and XML schema, occasionally ambiguities occur on what XML info set should be used to represent a method's return value or parameters. In order to remove these ambiguities, JAXB annotations may need to be generated on methods and method parameters to assure that the return value and the parameters are marshalled with the proper XML info set. A JAXB annotation on the method is used to specify the binding of a methods return type while an annotation on the parameter specifies the binding of that parameter. If the default XML info set for the return type or parameters correctly represents the XML info set, no JAXB annotations are needed.

Conformance (use of JAXB annotations): An SEI method MUST contain the appropriate JAXB annotations to assure that the proper XML info set is used when marshalling/unmarshalling the return type. Parameters of an SEI method MUST contain the appropriate JAXB annotations to assure that the

proper XML infoset is used when marshalling/unmarshalling the parameters of the method. The set of JAXB annotations that MUST be supported are: `javax.xml.bind.annotation.XmlAttachmentRef`, `javax.xml.bind.annotation.XmlList`, `javax.xml.bind.XmlMimeType` and `javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter`.

3.3.1.1. Non-wrapper Style

A `wsdl:message` is composed of zero or more `wsdl:part` elements. Message parts are classified as follows:

in

The message part is present only in the operation's input message.

out

The message part is present only in the operation's output message.

in/out

The message part is present in both the operation's input message and output message.

Two parts are considered equal if they have the same values for their `name` attribute and they reference the same global element or type. Using non-wrapper style, message parts are mapped to Java parameters according to their classification as follows:

in

The message part is mapped to a method parameter.

out

The message part is mapped to a method parameter using a holder class (see section [wsdl112jholders]) or is mapped to the method return type.

in/out

The message part is mapped to a method parameter using a holder class.

Conformance (Non-wrapped parameter naming): In the absence of any customizations, the name of a mapped Java method parameter MUST be the value of the `name` attribute of the `wsdl:part` element mapped according to the rules described in sections [xmlnames] and [namecollisions].

An application MAY customize this mapping using the `jaxws:parameter` binding declaration defined in section [bindingdeclporttypeoperation].

Section [paramorderandreturn] defines rules that govern the ordering of parameters in mapped Java methods and identification of the part that is mapped to the method return type.

3.3.1.2. Wrapper Style

A WSDL operation qualifies for wrapper style mapping only if the following criteria are met:

1. The operation's input and output messages (if present) each contain only a single part
2. The input message part refers to a global element declaration whose localname is equal to the

operation name

3. The output message (if present) part refers to a global element declaration
4. The elements referred to by the input and output message (if present) parts (henceforth referred to as *wrapper* elements) are both complex types defined using the `xsd:sequence` compositor
5. The wrapper elements only contain child elements, they MUST not contain other structures such as wildcards (element or attribute), `xsd:choice`, substitution groups (element references are not permitted) or attributes; furthermore, they MUST not be nillable.

Conformance (Default mapping mode): Operations that do not meet the criteria above MUST be mapped using non-wrapper style.

In some cases use of the wrapper style mapping can lead to undesirable Java method signatures and use of non-wrapper style mapping would be preferred.

Conformance (Disabling wrapper style): An implementation MUST support use of the `jaxws:enableWrapperStyle` binding declaration to enable or disable the wrapper style mapping of operations (see section [bindingdeclporttypeoperation]).

Using wrapper style, the child elements of the wrapper element (henceforth called *wrapper children*) are mapped to Java parameters, wrapper children are classified as follows:

in

The wrapper child is only present in the input message part's wrapper element.

out

The wrapper child is only present in the output message part's wrapper element.

in/out

The wrapper child is present in both the input and output message part's wrapper element.

Two wrapper children are considered equal if they have the same local name, the same XML schema type and the same Java type after mapping (see section [wsdl11typemapping] for XML Schema to Java type mapping rules). The mapping depends on the classification of the wrapper child as follows:

in

The wrapper child is mapped to a method parameter.

out

The wrapper child is mapped to a method parameter using a holder class (see section [wsd112jholders]) or is mapped to the method return value.

in/out

The wrapper child is mapped to a method parameter using a holder class.

Conformance (Wrapped parameter naming): In the absence of customization, the name of a mapped Java method parameter MUST be the value of the local name of the wrapper child mapped according to

the rules described in sections [xmlnames] and [namecollisions].

An application MAY customize this mapping using the `jaxws:parameter` binding declaration defined in section [bindingdeclporttypeoperation].

Conformance (Parameter name clash): If the mapping results in two Java parameters with the same name and one of those parameters is not mapped to the method return type, see section [paramorderandreturn], then this is reported as an error and requires developer intervention to correct, either by disabling wrapper style mapping, modifying the source WSDL or by specifying a customized parameter name mapping.

Conformance (Using `javax.xml.ws.RequestWrapper`): If wrapper style is used, generated Java methods MUST be annotated with a `javax.xml.ws.RequestWrapper` annotation. The annotation MAY be omitted if all its properties would have the default values.

Conformance (Using `javax.xml.ws.ResponseWrapper`): If wrapper style is used, generated Java methods MUST be annotated with a `javax.xml.ws.ResponseWrapper` annotation. The annotation MAY be omitted if all its properties would have the default values.

3.3.1.3. Example

Figure [w112jwrapperex] shows a WSDL extract and the Java method that results from using wrapper and non-wrapper mapping styles. For readability, annotations are omitted.

```

<!-- WSDL extract -->
<types>
  <xsd:element name="setLastTradePrice">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="tickerSymbol" type="xsd:string"/>
        <xsd:element name="lastTradePrice" type="xsd:float"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="setLastTradePriceResponse">
    <xsd:complexType>
      <xsd:sequence/>
    </xsd:complexType>
  </xsd:element>
</types>

<message name="setLastTradePrice">
  <part name="setLastTradePrice"
    element="tns:setLastTradePrice"/>
</message>

<message name="setLastTradePriceResponse">
  <part name="setLastTradePriceResponse"
    element="tns:setLastTradePriceResponse"/>
</message>

<portType name="StockQuoteUpdater">
  <operation name="setLastTradePrice">
    <input message="tns:setLastTradePrice"/>
    <output message="tns:setLastTradePriceResponse"/>
  </operation>
</portType>

// non-wrapper style mapping
SetLastTradePriceResponse setLastTradePrice(
  SetLastTradePrice setLastTradePrice);

// wrapper style mapping
void setLastTradePrice(String tickerSymbol, float lastTradePrice);

```

Figure 2.3: Wrapper and non-wrapper mapping styles

3.3.2. Parameter Order and Return Type

A `wsdl:operation` element may have a `parameterOrder` attribute that defines the ordering of parameters in a mapped Java method as follows:

- Message parts are either listed or unlisted. If the value of a `wsdl:part` element's `name` attribute is present in the `parameterOrder` attribute then the part is listed, otherwise it is unlisted.

Note: *R2305 in WS-I Basic Profile 1.1 (Ballinger, Ehnebuske, Ferris, et al. 2004) requires that if the `parameterOrder` attribute is present then at most one part may be unlisted. However, the algorithm outlined in this section supports WSDLs that do not conform with this requirement.*

- Parameters that are mapped from message parts are either listed or unlisted. Parameters that are mapped from listed parts are listed; parameters that are mapped from unlisted parts are unlisted.
- Parameters that are mapped from wrapper children (wrapper style mapping only) are unlisted.
- Listed parameters appear first in the method signature in the order in which their corresponding parts are listed in the `parameterOrder` attribute.
- Unlisted parameters either form the return type or follow the listed parameters
- The return type is determined as follows:

Non-wrapper style mapping

Only parameters that are mapped from parts in the abstract output message may form the return type, parts from other messages (see e.g. section [wsdl11headbindext]) do not qualify. If there is a single unlisted `out` part in the abstract output message then it forms the method return type, otherwise the return type is `void`.

Wrapper style mapping

If there is a single `out` wrapper child then it forms the method return type, if there is an `out` wrapper child with a local name of `return` then it forms the method return type, otherwise the return type is `void`.

- Unlisted parameters that do not form the return type follow the listed parameters in the following order:
 1. Parameters mapped from `in` and `in/out` parts appear in the same order the corresponding parts appear in the input message.
 2. Parameters mapped from `in` and `in/out` wrapper children (wrapper style mapping only) appear in the same order as the corresponding elements appear in the wrapper.
 3. Parameters mapped from `out` parts appear in the same order the corresponding parts appear in the output message.
 4. Parameters mapped from `out` wrapper children (wrapper style mapping only) appear in the same order as the corresponding wrapper children appear in the wrapper.

3.3.3. Holder Class

Holder classes are used to support **out** and **in/out** parameters in mapped method signatures. They provide a mutable wrapper for otherwise immutable object references. JAX-WS defines a generic holder class (`javax.xml.ws.Holder<T>`) that can be used for any Java class.

Parameters whose XML data type would normally be mapped to a Java primitive type (e.g., `xsd:int` to `int`) are instead mapped to a **Holder** whose type parameter is bound to the Java wrapper class corresponding to the primitive type. E.g., an **out** or **in/out** parameter whose XML data type would normally be mapped to a Java `int` is instead mapped to `Holder<java.lang.Integer>`.

*Conformance (Use of **Holder**):* Implementations **MUST** map **out** and **in/out** method parameters using `javax.xml.ws.Holder<T>`, with the exception of a **out** part that has been mapped to the method's return type.

3.3.4. Asynchrony

In addition to the synchronous mapping of `wsdl:operation` described above, a client side asynchronous mapping is also supported. It is expected that the asynchronous mapping will be useful in some but not all cases and therefore generation of the client side asynchronous methods should be optional at the users discretion.

Conformance (Asynchronous mapping required): An implementation **MUST** support the asynchronous mapping.

Conformance (Asynchronous mapping option): An implementation **MUST** support use of the `jaxws:enableAsyncMapping` binding declaration defined in section [PortType Operation](#) to enable and disable the asynchronous mapping.

JSR-181 currently does not define annotations that can be used to mark a method as being asynchronous.

3.3.4.1. Standard Asynchronous Interfaces

The following standard interfaces are used in the asynchronous operation mapping:

`javax.xml.ws.Response`

A generic interface that is used to group the results of a method invocation with the response context. `Response` extends `Future<T>` to provide asynchronous result polling capabilities.

`javax.xml.ws.AsyncHandler`

A generic interface that clients implement to receive results in an asynchronous callback.

3.3.4.2. Operation

Each `wsdl:operation` is mapped to two additional methods in the corresponding service endpoint interface:

Polling method

A polling method returns a typed `Response<'ResponseBean>` that may be polled using methods inherited from `Future<T>` to determine when the operation has completed and to retrieve the results. See below for further details on *ResponseBean*.

Callback method

A callback method takes an additional final parameter that is an instance of a typed `AsyncHandler<'ResponseBean>` and returns a wildcard `Future<?>` that may be polled to determine when the operation has completed. The object returned from `Future<?>.get()` has no standard type. Client code should not attempt to cast the object to any particular type as this will result in non-portable behavior.

Conformance (Asynchronous method naming): In the absence of customizations, the name of the polling and callback methods MUST be the value of the `name` attribute of the `wsdl:operation` suffixed with `Async` mapped according to the rules described in sections [XML Names](#) and [Name Collisions](#).

Conformance (Asynchronous parameter naming): The name of the method parameter for the callback handler MUST be `asyncHandler`. Parameter name collisions require user intervention to correct, see section [Name Collisions](#).

An application MAY customize this mapping using the `jaxws:method` binding declaration defined in section [PortType Operation](#).

Conformance (Failed method invocation): If there is any error prior to invocation of the operation, an implementation MUST throw a `WebServiceException` [1: Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation, see section [Faults](#).].

3.3.4.3. Message and Part

The asynchronous mapping supports both wrapper and non-wrapper mapping styles, but differs in how it maps `out` and `in/out` parts or wrapper children:

in

The part or wrapper child is mapped to a method parameter as described in section [Message and Part](#).

out

The part or wrapper child is mapped to a property of the response bean (see below).

in/out

The part or wrapper child is mapped to a method parameter (no holder class) and to a property of the response bean.

3.3.4.4. Response Bean

A response bean is a mapping of an operation's output message, it contains properties for each `out` and

`in/out` message part or wrapper child.

Conformance (Response bean naming): In the absence of customizations, the name of a response bean MUST be the value of the `name` attribute of the `wsdl:operation` suffixed with `Response` mapped according to the rules described in sections [XML Names](#) and [Name Collisions](#).

A response bean is mapped from a global element declaration following the rules described in section [Types](#). The global element declaration is formed as follows (in order of preference):

- If the operation's output message contains a single part and that part refers to a global element declaration then use the referenced global element.
- Synthesize a global element declaration of a complex type defined using the `xsd:sequence` compositor. Each output message part is mapped to a child of the synthesized element as follows:
 - Each global element referred to by an output part is added as a child of the sequence.
 - Each part that refers to a type is added as a child of the sequence by creating an element in no namespace whose localname is the value of the `name` attribute of the `wsdl:part` element and whose type is the value of the `type` attribute of the `wsdl:part` element

If the resulting response bean has only a single property then the bean wrapper should be discarded in method signatures. In this case, if the property is a Java primitive type then it is boxed using the Java wrapper type (e.g. `int` to `Integer`) to enable its use with `Response`.

3.3.4.5. Faults

Mapping of WSDL faults to service specific exceptions is identical for both asynchronous and synchronous cases, section [Fault](#) describes the mapping. However, mapped asynchronous methods do not throw service specific exceptions directly. Instead a `java.util.concurrent.ExecutionException` is thrown when a client attempts to retrieve the results of an asynchronous method invocation via the `Response.get` method.

Conformance (Asynchronous fault reporting): A WSDL fault that occurs during execution of an asynchronous method invocation MUST be mapped to a `java.util.concurrent.ExecutionException` thrown when the client calls `Response.get`.

`Response` is a static generic interface whose `get` method cannot throw service specific exceptions. Instead of throwing a service specific exception, a `Response` instance throws an `ExecutionException` whose cause is set to an instance of the service specific exception mapped from the corresponding WSDL fault.

Conformance (Asynchronous fault cause): An `ExecutionException` that is thrown by the `get` method of `Response` as a result of a WSDL fault MUST have as its cause the service specific exception mapped from the WSDL fault, if there is one, otherwise the `ProtocolException` mapped from the WSDL fault (see [Exceptions](#)).

3.3.4.6. Mapping Examples

Figure [2.4] shows an example of the asynchronous operation mapping. Note that the mapping uses `Float` instead of a response bean wrapper (`GetPriceResponse`) since the synthesized global element declaration for the operations output message (lines 17–24) maps to a response bean that contains only a single property.

```
<!-- WSDL extract -->
<message name="getPrice">
  <part name="ticker" type="xsd:string"/>
</message>

<message name="getPriceResponse">
  <part name="price" type="xsd:float"/>
</message>

<portType name="StockQuote">
  <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
  </operation>
</portType>

<!-- Synthesized response bean element -->
<xsd:element name="getPriceResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="price" type="xsd:float"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

// synchronous mapping
@WebService
public interface StockQuote {
  float getPrice(String ticker);
}

// asynchronous mapping
@WebService
public interface StockQuote {
  float getPrice(String ticker);
  Response<Float> getPriceAsync(String ticker);
  Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
}
```

Figure 2.4: Asynchronous operation mapping

3.3.4.7. Usage Examples

- Synchronous use.

```
Service service = ...;
StockQuote quoteService = (StockQuote)service.getPort(portName);
Float quote = quoteService.getPrice(ticker);
```

- Asynchronous polling use.

```
Service service = ...;
StockQuote quoteService = (StockQuote)service.getPort(portName);
Response<Float> response = quoteService.getPriceAsync(ticker);
while (!response.isDone()) {
    // do something while we wait
}
Float quote = response.get();
```

- Asynchronous callback use.

```
class MyPriceHandler implements AsyncHandler<Float> {
    ...
    public void handleResponse(Response<Float> response) {
```

3.4. Types

Mapping of XML Schema types to Java is described by the JAXB 2.1 specification[\[11\]](#). The contents of a `wsdl:types` section is passed to JAXB along with any additional type or element declarations (e.g., see section [Asynchrony](#)) required to map other WSDL constructs to Java. E.g., section [Asynchrony](#) defines an algorithm for synthesizing additional global element declarations to provide a mapping from WSDL operations to asynchronous Java method signatures.

JAXB supports mapping XML types to either Java interfaces or classes. By default JAX-WS uses the class based mapping of JAXB but also allows use of the interface based mapping.

Conformance (JAXB class mapping): In the absence of user customizations, an implementation **MUST** use the JAXB class based mapping with `generateValueClass` set to `true` and `generateElementClass` set to `false` when mapping WSDL types to Java.

Conformance (JAXB customization use): An implementation **MUST** support use of JAXB customizations during mapping as detailed in section [Using JAXB Binding Declarations](#).

Conformance (JAXB customization clash): To avoid clashes, if a user customizes the mapping, an implementation MUST NOT add the default class based mapping customizations.

In addition, for ease of use, JAX-WS strips any `JAXBElement<T>` wrapper off the type of a method parameter if the normal JAXB mapping would result in one [2: JAXB maps an element declaration to a Java instance that implements `JAXBElement`]. E.g. a parameter that JAXB would map to `JAXBElement<Integer>` is instead be mapped to `Integer`.

JAXB provides support for the SOAP MTOM[30] /XOP[31] mechanism for optimizing transmission of binary data types. JAX-WS provides the MIME processing required to enable JAXB to serialize and deserialize MIME based MTOM/XOP packages. The contract between JAXB and an MTOM/XOP package processor is defined by the `javax.xml.bind.AttachmentMarshaller` and `javax.xml.bind.AttachmentUnmarshaller` classes. A JAX-WS implementation can plug into it by registering its own `AttachmentMarshaller` and `AttachmentUnmarshaller` at runtime using the `setAttachmentUnmarshaller` method of `javax.xml.bind.Unmarshaller` (resp. the `setAttachmentMarshaller` method of `javax.xml.bind.Marshaller`).

3.4.1. W3CEndpointReference

JAXB 2.1 by default does not map `wsa:EndpointReference` to the `javax.xml.ws.wsaddressing.W3CEndpointReference` class. However, for JAX-WS developers to fully utilize the use of a `wsa:EndpointReference`, JAX-WS implementations MUST map the `wsa:EndpointReference`

and its subtypes to `javax.xml.ws.W3CEndpointReference` by default. JAXB 2.1 provides a standard customization that can be used to cause this mapping. Implementations may provide a way to map these types differently.

Conformance (javax.xml.ws.wsaddressing.W3CEndpointReference): Any schema element of the type `wsa:EndpointReference` or its subtypes MUST be mapped to `javax.xml.ws.wsaddressing.W3CEndpointReference` by default.

3.5. Fault

A `wsdl:fault` element is mapped to a Java exception.

Conformance (javax.xml.ws.WebFault required): A mapped exception MUST be annotated with a `javax.xml.ws.WebFault` annotation.

Conformance (Exception naming): In the absence of customizations, the name of a mapped exception MUST be the value of the `name` attribute of the `wsdl:message` referred to by the `wsdl:fault` element mapped according to the rules in sections [XML Names](#) and [Name Collisions](#).

An application MAY customize this mapping using the `jaxws:class` binding declaration defined in section [PortType Fault Message](#).

Multiple operations within the same service can define equivalent faults. Faults defined within the

same service are equivalent if the values of their `message` attributes are equal.

Conformance (Fault equivalence): An implementation MUST map equivalent faults within a service to a single Java exception class.

A `wsdl:fault` element refers to a `wsdl:message` that contains a single part. The global element declaration [3: WS-I Basic Profile[20] R2205 requires parts to refer to elements rather than types.] referred to by that part is mapped to a Java bean, henceforth called a *fault bean*, using the mapping described in section [Types](#). An implementation generates a wrapper exception class that extends `java.lang.Exception` and contains the following methods:

WrapperException`(**String** `message`, *FaultBean*``faultInfo`)

A constructor where *WrapperException* is replaced with the name of the generated wrapper exception and *FaultBean* is replaced by the name of the generated fault bean.

WrapperException`(**String** `message`, *FaultBean*``faultInfo`, **Throwable** `cause`)

A constructor where *WrapperException* is replaced with the name of the generated wrapper exception and *FaultBean* is replaced by the name of the generated fault bean. The last argument, `cause`, may be used to convey protocol specific fault information, see section [Protocol Specific Exception Handling](#).

FaultBean`**getFaultInfo()**`

Getter to obtain the fault information, where *FaultBean* is replaced by the name of the generated fault bean.

The *WrapperException* class is annotated using the `WebFault` annotation (see section [javax.xml.ws.WebFault](#)) to capture the local and namespace name of the global element mapped to the fault bean.

Two `wsdl:fault` child elements of the same `wsdl:operation` that indirectly refer to the same global element declaration are considered to be equivalent since there is no interoperable way of differentiating between their serialized forms.

Conformance (Fault equivalence): At runtime an implementation MAY map a serialized fault into any equivalent Java exception.

3.5.1. Example

Figure [\[2.5\]](#) shows an example of the WSDL fault mapping described above.

```

<!-- WSDL extract -->
<types>
  <xsd:schema targetNamespace="...">
    <xsd:element name="faultDetail">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="majorCode" type="xsd:int"/>
          <xsd:element name="minorCode" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</types>

<message name="operationException">
  <part name="faultDetail" element="tns:faultDetail"/>
</message>

<portType name="StockQuoteUpdater">
  <operation name="setLastTradePrice">
    <input .../>
    <output .../>
    <fault name="operationException"
      message="tns:operationException"/>
  </operation>
</portType>

// fault mapping
@WebFault(name="faultDetail", targetNamespace="...")
class OperationException extends Exception {
  OperationException(String message, FaultDetail faultInfo) {...}
  OperationException(String message, FaultDetail faultInfo,
    Throwable cause) {...}
  FaultDetail getFaultInfo() {...}
}

```

Figure 2.5: Fault mapping

3.6. Binding

The mapping from WSDL 1.1 to Java is based on the abstract description of a `wsdl:portType` and its associated operations. However, the binding of a port type to a protocol can introduce changes in the mapping – this section describes those changes in the general case and specifically for the mandatory WSDL 1.1 protocol bindings.

Conformance (Required WSDL extensions): An implementation MUST support mapping of the WSDL 1.1 specified extension elements for the WSDL SOAP and MIME bindings.

3.6.1.

General Considerations

R2209 in WS-I Simple SOAP Binding Profile 1.1[32] recommends that all parts of a message be bound but does not require it.

Conformance (Unbound message parts): To preserve the protocol independence of mapped operations, an implementation MUST NOT ignore unbound message parts when mapping from WSDL 1.1 to Java. Instead an implementation MUST generate binding code that ignores **in** and **in/out** parameters mapped from unbound parts and that presents **out** parameters mapped from unbound parts as **null**.

3.6.2. SOAP Binding

This section describes changes to the WSDL 1.1 to Java mapping that may result from use of certain SOAP binding extensions.

3.6.2.1. Header Binding Extension

A **soap:header** element may be used to bind a part from a message to a SOAP header. As clarified by R2208 in WS-I Basic Profile 1.1[20], the part may belong to either the message bound by the **soap:body** or to a different message:

- If the part belongs to the message bound by the **soap:body** then it is mapped to a method parameter as described in section [Operation](#). Such a part is always mapped using the non-wrapper style.
- If the part belongs to a different message than that bound by the **soap:body** then it may optionally be mapped to an additional method parameter. When mapped to a parameter, the part is treated as an additional unlisted part for the purposes of the mapping described in section [Operation](#). This additional part does not affect eligibility for wrapper style mapping of the message bound by the **soap:body** (see section [Message and Part](#)); the additional part is always mapped using the non-wrapper style.

Note that the order of headers in a SOAP message is independent of the order of **soap:header** elements in the WSDL binding – see R2751 in WS-I Basic Profile 1.0[8]. This causes problems when two or more headers with the same qualified name are present in a message and one or more of those headers are bound to a method parameter since it is not possible to determine which header maps to which parameter.

Conformance (Duplicate headers in binding): When mapping, an implementation MUST report an error if the binding of an operation includes two or more **soap:header** elements that would result in SOAP headers with the same qualified name.

Conformance (Duplicate headers in message): An implementation MUST generate a runtime error if, during unmarshalling, there is more than one instance of a header whose qualified name is mapped to a method parameter.

3.6.3. MIME Binding

The presence of a `mime:multipartRelated` binding extension element as a child of a `wsdl:input` or `wsdl:output` element in a `wsdl:binding` indicates that the corresponding messages may be serialized as MIME packages. The WS-I Attachments Profile[33] describes two separate attachment mechanisms, both based on use of the WSDL 1.1 MIME binding[5]:

`wsiap:swaRef`

A schema type that may be used in the abstract message description to indicate a reference to an attachment.

`mime:content`

A binding construct that may be used to bind a message part to an attachment.

JAXB[10] describes the mapping from the WS-I defined `wsiap:swaRef` schema type to Java and, since JAX-WS inherits this capability, it is not discussed further here. Use of the `mime:content` construct is outside the scope of JAXB mapping and the following subsection describes changes to the WSDL 1.1 to Java mapping that results from its use.

3.6.3.1. `mime:content`

Message parts are mapped to method parameters as described in section [Operation](#) regardless of whether the part is bound to the SOAP message or to an attachment. JAXB rules are used to determine the Java type of message parts based on the XML schema type referenced by the `wsdl:part`. However, when a message part is bound to a MIME part (using the `mime:content` element of the WSDL MIME binding) additional information is available that provides the MIME type of the data and this can optionally be used to narrow the default JAXB mapping. This use of additional metadata in `mime:content` elements is disabled by default for WSDL to Java mapping, but can be enabled using `jaxws:enableMIMEContent` customization (section [Binding](#)).

Conformance (Use of MIME type information): An implementation MUST support using the `jaxws:enableMIMEContent` binding declaration defined in section [Binding](#) to enable or disable the use of the additional metadata in `mime:content` elements when mapping from WSDL to Java.

JAXB defines a mapping between MIME types and Java types. When a part is bound using one or more `mime:content` elements [4: Multiple `mime:content` elements for the same part indicate a set of permissible alternate types.] and use of the additional metadata is enabled then the JAXB mapping is customized to use the most specific type allowed by the set of MIME types described for the part in the binding. The case where the parameter mode is `INOUT` and is bound to different mime bindings in the input and output messages using the `mime:content` element MUST also be treated in the same way as described above. Please refer to appendix H in the JAXB 2.0 specification [10] for details of the type mapping.

The part belongs to the message bound by the `soap:body` then it is mapped to a method parameter as described in section [Operation](#). Such a part is always mapped using the non-wrapper style.

Parts bound to MIME using the `mime:content` WSDL extension are mapped as described in section [Operation](#). These parts are mapped using the non-wrapper style.

Figure [\[2.6\]](#) shows an example WSDL and two mapped interfaces: one without using the `mime:content` metadata, the other using the additional metadata to narrow the binding. Note that in the latter the type of the `claimPhoto` method parameter is `Image` rather than the default `byte[]`.

```

<!-- WSDL extract -->
<wsdl:message name="ClaimIn">
  <wsdl:part name="body" element="types:ClaimDetail"/>
  <wsdl:part name="ClaimPhoto" type="xsd:base64Binary"/>
</wsdl:message>

<wsdl:portType name="ClaimPortType">
  <wsdl:operation name="SendClaim">
    <wsdl:input message="tns:ClaimIn"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ClaimBinding" type="tns:ClaimPortType">
  <soapbind:binding style="document" transport="..."/>
  <wsdl:operation name="SendClaim">
    <soapbind:operation soapAction="..."/>
    <wsdl:input>
      <mime:multipartRelated>
        <mime:part>
          <soapbind:body parts="body" use="literal"/>
        </mime:part>
        <mime:part>
          <mime:content part="ClaimPhoto" type="image/jpeg"/>
          <mime:content part="ClaimPhoto" type="image/gif"/>
        </mime:part>
      </mime:multipartRelated>
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>

// Mapped Java interface without mime:content metadata
@WebService
public interface ClaimPortType {
  public String sendClaim(ClaimDetail detail, byte claimPhoto[]);
}

// Mapped Java interface using mime:content metadata
@WebService
public interface ClaimPortType {
  public String sendClaim(ClaimDetail detail, Image claimPhoto);
}

```

Figure 2.6: Use of `mime:content` metadata

Conformance (MIME type mismatch): On receipt of a message where the MIME type of a part does not match that described in the WSDL an implementation SHOULD throw a `WebServiceException`.

Conformance (MIME part identification): An implementation MUST use the algorithm defined in the WS-I Attachments Profile[33] when generating the MIME `Content-ID` header field value for a part bound using `mime:content`.

3.7. Service and Port

A `wsdl:service` is a collection of related `wsdl:port` elements. A `wsdl:port` element describes a port type bound to a particular protocol (a `wsdl:binding`) that is available at particular endpoint address. On the client side, a `wsdl:service` element is mapped to a generated service class that extends `javax.xml.ws.Service` (see section [javax.xml.ws.Service](#) for more information on the `Service` class).

Conformance (Service superclass required): A generated service class MUST extend the `javax.xml.ws.Service` class.

Conformance (Service class naming): In the absence of customization, the name of a generated service class MUST be the value of the `name` attribute of the `wsdl:service` element mapped according to the rules described in sections [XML Names](#) and [Name Collisions](#).

An application MAY customize the name of the generated service class using the `jaxws:class` binding declaration defined in section [Service](#).

In order to allow an implementation to identify the Web service that a generated service class corresponds to, the latter is required to be annotated with `javax.xml.ws.WebServiceClient` annotation. The annotation contains all the information necessary to locate a WSDL document and uniquely identify a `wsdl:service` inside it.

Conformance (javax.xml.ws.WebServiceClient required): A generated service class MUST be annotated with a `javax.xml.ws.WebServiceClient` annotation.

JAX-WS 2.2 mandates that six constructors be present on every generated service class.

Conformance (Generated service default constructor): A generated service class MUST have a default (i.e. zero-argument) public constructor. This constructor MUST call the `javax.xml.ws.Service(URL, QName)` protected constructor, passing as arguments the WSDL location and the service name. The values of the actual arguments for this call MUST be equal (in the `java.lang.Object.equals` sense) to the values specified in the mandatory `WebServiceClient` annotation on the generated service class itself.

Conformance (Generated service (WebServiceFeature ...) constructor): A generated service class MUST have a public constructor that takes one argument, the web service features (a varargs `javax.xml.ws.WebServiceFeature`). This constructor MUST call the `javax.xml.ws.Service(URL, QName, WebServiceFeature ...)` protected constructor, passing as arguments the WSDL location, the service name and the web service features. The values of the actual arguments WSDL location, service name for this call are as specified in the mandatory `WebServiceClient` annotation on the generated service class itself, and the value of the web service features argument is with which it was invoked.

Conformance (Generated service (URL) constructor): The implementation class MUST have a public

constructor that takes one argument, the WSDL location (a `java.net.URL`). This constructor MUST call the `javax.xml.ws.Service(URL, QName)` protected constructor. The values of the actual arguments WSDL location is with which it was invoked and the service name is as specified in the mandatory `WebServiceClient` annotation on the generated service class itself.

Conformance (Generated service (URL,WebServiceFeature...) constructor): The implementation class MUST have a public constructor that takes two arguments, the WSDL location (a `java.net.URL`) and the web service features (a varargs `javax.xml.ws.WebServiceFeature`). This constructor MUST call the `javax.xml.ws.Service(URL, QName, WebServiceFeature ...)` protected constructor, passing as arguments the WSDL location, the service name and the web service features. The values of the actual arguments WSDL location and the web services features are with which it was invoked and the service name is as specified in the mandatory `WebServiceClient` annotation on the generated service class itself.

Conformance (Generated service (URL,QName) constructor): The implementation class MUST have a public constructor that takes two arguments, the WSDL location (a `java.net.URL`) and the service name (a `javax.xml.namespace.QName`). This constructor MUST call the `javax.xml.ws.Service(URL, QName)` protected constructor, passing as arguments the WSDL location and the service name values with which it was invoked.

Conformance (Generated service (URL,QName,WebServiceFeature...) constructor): The implementation class MUST have a public constructor that takes three arguments, the WSDL location (a `java.net.URL`), the service name (a `javax.xml.namespace.QName`) and the web service features (a varargs `javax.xml.ws.WebServiceFeature`). This constructor MUST call the `javax.xml.ws.Service(URL, QName, WebServiceFeature ...)` protected constructor, passing as arguments the WSDL location, the service name and the web service feature values with which it was invoked.

For each port in the service, the generated client side service class contains the following methods, two for each port defined by the WSDL service and whose binding is supported by the JAX-WS implementation:

`getPortName()`

One required method that takes no parameters and returns a proxy that implements the mapped service endpoint interface. The method generated delegates to the `Service.getPort(...)` method passing it the port name. The value of the port name MUST be equal to the value specified in the mandatory `WebEndpoint` annotation on the method itself.

`getPortName(WebServiceFeature... features)`

One required method that takes a variable-length array of `javax.xml.ws.WebServiceFeature` and returns a proxy that implements the mapped service endpoint interface. The method generated delegates to the `Service.getPort(QName portName, Class<T> SEI, WebServiceFeature... features)` method passing it the port name, the SEI and the features. The value of the port name MUST be equal to the value specified in the mandatory `WebEndpoint` annotation on the method itself.

Conformance (Failed getPort Method): A generated `getPortName` method MUST throw `javax.xml.ws.WebServiceException` on failure.

The value of *PortName* in the above is derived as follows: the value of the `name` attribute of the `wsdl:port` element is first mapped to a Java identifier according to the rules described in section [XML Names](#), this Java identifier is then treated as a JavaBean property for the purposes of deriving the `get`PortName` method name.

An application MAY customize the name of the generated methods for a port using the `jaxws:method` binding declaration defined in section [Port](#).

In order to enable an implementation to determine the `wsdl:port` that a port getter method corresponds to, the latter is required to be annotated with a `javax.xml.ws.WebEndpoint` annotation.

Conformance (javax.xml.ws.WebEndpoint required): The `get`PortName` methods of generated service interface MUST be annotated with a `javax.xml.ws.WebEndpoint` annotation.

3.7.1. Example

The following shows a WSDL extract and the resulting generated service class.

```
<!-- WSDL extract -->
<wsdl:service name="StockQuoteService">
  <wsdl:port name="StockQuoteHTTPPort" binding="StockQuoteHTTPBinding"/>
  <wsdl:port name="StockQuoteSMTPPort" binding="StockQuoteSMTPBinding"/>
</wsdl:service>

// Generated Service Class
@WebServiceClient(name="StockQuoteService",
    targetNamespace="http://example.com/stocks",
    wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service {

    public StockQuoteService() {
        super(new URL("http://example.com/stocks.wsdl"),
            new QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(WebServiceFeature ... features) {
        super(new URL("http://example.com/stocks.wsdl"),
            new QName("http://example.com/stocks", "StockQuoteService"),
            features);
    }

    public StockQuoteService(URL wsdlLocation) {
        super(wsdlLocation,
            new QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(URL wsdlLocation, WebServiceFeature ... features) {
```



```

        super(wsdlLocation,
            new QName("http://example.com/stocks", "StockQuoteService"),
            features);
    }

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

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

    @WebEndpoint(name="StockQuoteHTTPPort")
    public StockQuoteProvider getStockQuoteHTTPPort() {
        return (StockQuoteProvider)super.getPort(
            new QName("http://example.com/stocks", "StockQuoteHTTPPort"),
            StockQuoteProvider.class);
    }

    @WebEndpoint(name="StockQuoteHTTPPort")
    public StockQuoteProvider getStockQuoteHTTPPort(WebServiceFeature... features) {
        return (StockQuoteProvider)super.getPort(
            new QName("http://example.com/stocks", "StockQuoteHTTPPort"),
            StockQuoteProvider.class,
            features);
    }

    @WebEndpoint(name="StockQuoteSMTPPort")
    public StockQuoteProvider getStockQuoteSMTPPort() {
        return (StockQuoteProvider)super.getPort(
            new QName("http://example.com/stocks", "StockQuoteSMTPPort"),
            StockQuoteProvider.class);
    }

    @WebEndpoint(name="StockQuoteSMTPPort")
    public StockQuoteProvider getStockQuoteSMTPPort(WebServiceFeature... features) {
        return (StockQuoteProvider)super.getPort(
            new QName("http://example.com/stocks", "StockQuoteSMTPPort"),
            StockQuoteProvider.class,
            features);
    }
}

```

In the above, `StockQuoteProvider` is the service endpoint interface mapped from the WSDL port type for both referenced bindings.

3.8. XML Names

Appendix C of JAXB 1.0[\[9\]](#) defines a mapping from XML names to Java identifiers. JAX-WS uses this mapping to convert WSDL identifiers to Java identifiers with the following modifications and additions:

Method identifiers

When mapping `wsdl:operation` names to Java method identifiers, the `get` or `set` prefix is not added. Instead the first word in the word-list has its first character converted to lower case.

Parameter identifiers

When mapping `wsdl:part` names or wrapper child local names to Java method parameter identifiers, the first word in the word-list has its first character converted to lower case. Clashes with Java language reserved words are reported as errors and require use of appropriate customizations to fix the clash.

3.8.1. Name Collisions

WSDL name scoping rules may result in name collisions when mapping from WSDL 1.1 to Java. E.g., a port type and a service are both mapped to Java classes but WSDL allows both to be given the same name. This section defines rules for resolving such name collisions.

The order of precedence for name collision resolution is as follows (highest to lowest);

1. Service endpoint interface
2. Non-exception Java class
3. Exception class
4. Service class

If a name collision occurs between two identifiers with different precedences, the lower precedence item has its name changed as follows:

Non-exception Java class

The suffix `_Type` is added to the class name.

Exception class

The suffix `_Exception` is added to the class name.

Service class

The suffix `_Service` is added to the class name.

If a name collision occurs between two identifiers with the same precedence, this is reported as an error and requires developer intervention to correct. The error may be corrected either by modifying the source WSDL or by specifying a customized name mapping.

If a name collision occurs between a mapped Java method and a method in `javax.xml.ws.BindingProvider` (an interface that proxies are required to implement, see section [javax.xml.ws.BindingProvider](#)), the prefix `_` is added to the mapped method.

Chapter 4. Java to WSDL 1.1 Mapping

This chapter describes the mapping from Java to WSDL 1.1. This mapping is used when generating web service endpoints from existing Java interfaces.

Conformance (WSDL 1.1 support): Implementations **MUST** support mapping Java to WSDL 1.1.

The following sections describe the default mapping from each Java construct to the equivalent WSDL 1.1 artifact.

An application **MAY** customize the mapping using the annotations defined in section [Annotations](#).

Conformance (Standard annotations): An implementation **MUST** support the use of annotations defined in section [metadata] to customize the Java to WSDL 1.1 mapping.

4.1. Java Names

Conformance (Java identifier mapping): In the absence of annotations described in this specification, Java identifiers **MUST** be mapped to XML names using the algorithm defined in appendix B of SOAP 1.2 Part 2[4].

4.1.1. Name Collisions

WS-I Basic Profile 1.0[8] (see R2304) requires operations within a `wsdl:portType` to be uniquely named – support for customization of the operation name allows this requirement to be met when a Java SEI contains overloaded methods.

Conformance (Method name disambiguation): An implementation **MUST** support the use of the `javax.jws.WebMethod` annotation to disambiguate overloaded Java method names when mapped to WSDL.

4.2. Package

A Java package is mapped to a `wsdl:definitions` element and an associated `targetNamespace` attribute. The `wsdl:definitions` element acts as a container for other WSDL elements that together form the WSDL description of the constructs in the corresponding Java package.

A default value for the `targetNamespace` attribute is derived from the package name as follows:

1. The package name is tokenized using the `.` character as a delimiter.
2. The order of the tokens is reversed.
3. The value of the `targetNamespace` attribute is obtained by concatenating `http://to` the list of tokens separated by `.` and `/`.

E.g., the Java package `com.example.ws` would be mapped to the target namespace

<http://ws.example.com/>.

Conformance (Package name mapping): The `javax.jws.WebService` annotation (see section [javax.jws.WebService](#)) MAY be used to specify the target namespace to use for a Web service and MUST be used for classes or interfaces in no package. In the absence of a `javax.jws.WebService` annotation the Java package name MUST be mapped to the value of the `wsdl:definitions` element's `targetNamespace` attribute using the algorithm defined above.

No specific authoring style is required for the mapped WSDL document; implementations are free to generate WSDL that uses the WSDL and XML Schema import directives.

Conformance (WSDL and XML Schema import directives): Generated WSDL MUST comply with the WS-I Basic Profile 1.0[8] restrictions (See R2001, R2002, and R2003) on usage of WSDL and XML Schema import directives.

4.3. Class

A Java class (not an interface) annotated with a `javax.jws.WebService` annotation can be used to define a Web service.

In order to allow for a separation between Web service interface and implementation, if the `WebService` annotation on the class under consideration has a `endpointInterface` element, then the interface referred by this element is for all purposes the SEI associated with the class.

Otherwise, the class implicitly defines a service endpoint interface (SEI) which comprises all of the public

non-static or non-final

methods that satisfy one of the following conditions:

1. They are annotated with the `javax.jws.WebMethod` annotation with the `exclude` element set to `false` or missing (since `false` is the default for this annotation element).
2. They are not annotated with the `javax.jws.WebMethod` annotation but their declaring class has a `javax.jws.WebService` annotation.

For mapping purposes, this implicit SEI and its methods are considered to be annotated with the same Web service-related annotations that the original class and its methods have.

In practice, in order to exclude a public method of a class annotated with `WebService` and not directly specifying a `endpointInterface` from the implicitly defined SEI, it is necessary to annotate the method with a `WebMethod` annotation with the `exclude` element set to `true`.

Conformance (Class mapping): An implementation MUST support the mapping of `javax.jws.WebService` annotated classes to implicit service endpoint interfaces.

For mapping purposes, this class must be a top level class or a static inner class. As defined by JSR 181,

a class annotated with `javax.jws.WebService` must have a default public constructor.

4.4. Interface

A Java service endpoint interface (SEI) is mapped to a `wsdl:portType` element. The `wsdl:portType` element acts as a container for other WSDL elements that together form the WSDL description of the methods in the corresponding Java SEI. An SEI is a Java interface that meets all of the following criteria:

- It MUST carry a `javax.jws.WebService` annotation (see [javax.jws.WebService](#)).
- Any of its methods MAY carry a `javax.jws.WebMethod` annotation (see [javax.jws.WebMethod](#)).
- `javax.jws.WebMethod` if used, MUST NOT have the `exclude` element set to `true`.
- All method parameters and return types are compatible with the JAXB 2.0[10] Java to XML Schema mapping definition

Conformance (portType naming): The `javax.jws.WebService` annotation (see section [javax.jws.WebService](#)) MAY be used to customize the `name` and `targetNamespace` attributes of the `wsdl:portType` element. If not customized, the value of the `name` attribute of the `wsdl:portType` element MUST be the name of the SEI not including the package name and the target namespace is computed as defined above in section [Package](#).

Figure [\[3.1\]](#) shows an example of a Java SEI and the corresponding `wsdl:portType`.

Multiple SEIs in the same package may result in name clashes as the result of sections [Document Wrapped](#) and [Service Specific Exception](#) of the specification. Customizations may be used to resolve these clashes. See sections [javax.xml.ws.WebFault](#), [javax.xml.ws.RequestWrapper](#) and [javax.xml.ws.ResponseWrapper](#) for more information on these customizations.

4.4.1. Inheritance

WSDL 1.1 does not define a standard representation for the inheritance of `wsdl:portType` elements. When mapping an SEI that inherits from another interface, the SEI is treated as if all methods of the inherited interface were defined within the SEI.

Conformance (Inheritance flattening): A mapped `wsdl:portType` element MUST contain WSDL definitions for all the methods of the corresponding Java SEI including all inherited methods.

Conformance (Inherited interface mapping): An implementation MAY map inherited interfaces to additional `wsdl:portType` elements within the `wsdl:definitions` element.

4.5. Method

Each public method in a Java SEI is mapped to a `wsdl:operation` element in the corresponding `wsdl:portType` plus one or more `wsdl:message` elements.

Conformance (Operation naming): In the absence of customizations, the value of the `name` attribute of the `wsdl:operation` element MUST be the name of the Java method. The `javax.jws.WebMethod` (see [mdwebmethod]) annotation MAY be used to customize the value of the `name` attribute of the `wsdl:operation` element and MUST be used to resolve naming conflicts. If the `exclude` element of the `javax.jws.WebMethod` is set to `true` then the Java method MUST NOT be present in the wsdl as a `wsdl:operation` element.

Methods are either one-way or two-way: one way methods have an input but produce no output, two way methods have an input and produce an output. Section [j2wsdl11onewayops] describes one way operations further.

The `wsdl:operation` element corresponding to each method has one or more child elements as follows:

- A `wsdl:input` element that refers to an associated `wsdl:message` element to describe the operation input.
- (Two-way methods only) an optional `wsdl:output` element that refers to a `wsdl:message` to describe the operation output.
- (Two-way methods only) zero or more `wsdl:fault` child elements, one for each exception thrown by the method. The `wsdl:fault` child elements refer to associated `wsdl:message` elements to describe each fault. See section [j2w11except] for further details on exception mapping.
- `wsdl:input`, `wsdl:output`(if any), and `wsdl:fault`(if any) child elements must have `wsam:Action` attribute to describe WS-Addressing Action property. The value of the `wsam:Action` attribute is computed using the algorithm in section [wsam:Action Computation Algorithm](#)

The value of a `wsdl:message` element's `name` attribute is not significant but by convention it is normally equal to the corresponding operation name for input messages and the operation name concatenated with Response for output messages. Naming of fault messages is described in section [Service Specific Exception](#).

Conformance (Generating `wsam:Action`): `wsdl:operation`'s child elements `wsdl:input`, `wsdl:output`, and `wsdl:fault` MUST have the `wsam:Action` attribute. `wsam:Action` attribute MUST be computed using the algorithm that is specified in [wsam:Action Computation Algorithm](#)

Each `wsdl:message` element has one of the following [5: The `javax.jws.WebParam` and `javax.jws.WebResult` annotations can introduce additional parts into messages when the `header` element is `true`.]:

Document style

A single `wsdl:part` child element that refers, via an `element` attribute, to a global element declaration in the `wsdl:types` section.

RPC style

Zero or more `wsdl:part` child elements (one per method parameter and one for a non-void return value) that refer, via a `type` attribute, to named type declarations in the `wsdl:types` section.

Figure [3.1] shows an example of mapping a Java interface containing a single method to WSDL 1.1

using document style. Figure [3.2] shows an example of mapping a Java interface containing a single method to WSDL 1.1 using RPC style.

Section [Method Parameters and Return Type](#) describes the mapping from Java methods and their parameters to corresponding global element declarations and named types in the `wsdl:types` section.

```
// Java
package com.example;
@WebService
public interface StockQuoteProvider {
    float getPrice(String tickerSymbol)
        throws TickerException;
}

<!-- WSDL extract -->
<types>
  <xsd:schema targetNamespace="...">
    <!-- element declarations -->
    <xsd:element name="getPrice"
      type="tns:getPriceType"/>
    <xsd:element name="getPriceResponse"
      type="tns:getPriceResponseType"/>
    <xsd:element name="TickerException"
      type="tns:TickerExceptionType"/>

    <!-- type definitions -->
    ...
  </xsd:schema>
</types>

<message name="getPrice">
  <part name="getPrice" element="tns:getPrice"/>
</message>

<message name="getPriceResponse">
  <part name="getPriceResponse" element="tns:getPriceResponse"/>
</message>

<message name="TickerException">
  <part name="TickerException" element="tns:TickerException"/>
</message>

<portType name="StockQuoteProvider">
  <operation name="getPrice">
    <input message="tns:getPrice" wsam:action="..."/>
    <output message="tns:getPriceResponse" wsam:action="..."/>
    <fault message="tns:TickerException" wsam:action="..."/>
  </operation>
</portType>
```

Figure 3.1: Java interface to WSDL portType mapping using document style


```
// Java
package com.example;
@WebService
public interface StockQuoteProvider {
    float getPrice(String tickerSymbol)
        throws TickerException;
}

<!-- WSDL extract -->
<types>
  <xsd:schema targetNamespace="...">
    <!-- element declarations -->
    <xsd:element name="TickerException"
      type="tns:TickerExceptionType"/>

    <!-- type definitions -->
    ...
  </xsd:schema>
</types>

<message name="getPrice">
  <part name="tickerSymbol" type="xsd:string"/>
</message>

<message name="getPriceResponse">
  <part name="return" type="xsd:float"/>
</message>

<message name="TickerException">
  <part name="TickerException" element="tns:TickerException"/>
</message>

<portType name="StockQuoteProvider">
  <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
    <fault message="tns:TickerException"/>
  </operation>
</portType>
```

Figure 3.2: Java interface to WSDL portType mapping using RPC style

4.5.1. One Way Operations

Only Java methods whose return type is `void`, that have no parameters that implement `Holder` and that do not throw any checked exceptions can be mapped to one-way operations. Not all Java methods that

fulfill this requirement are amenable to become one-way operations and automatic choice between two-way and one-way mapping is not possible.

Conformance (One-way mapping): Implementations MUST support use of the `javax.jws.OneWay` (see [mdoneway]) annotation to specify which methods to map to one-way operations. Methods that are not annotated with `javax.jws.OneWay` MUST NOT be mapped to one-way operations.

Conformance (One-way mapping errors): Implementations MUST prevent mapping to one-way operations of methods that do not meet the necessary criteria.

4.5.2. wsam:Action Computation Algorithm

All `wsdl:operation`'s child elements `wsdl:input`, `wsdl:output` and `wsdl:fault` must have the `wsam:Action` attribute in the the generated WSDL. The algorithm to compute `wsam:Action` from SEI method is as follows:

1. A non-default `@Action(input=...)` or `@WebMethod(action=...)` value on a SEI method MUST result into `wsdl:input[@wsam:Action]` attribute in the corresponding `wsdl:operation`. Also, `@Action(input=...)` and `@WebMethod(action=...)` annotation element values MUST be same, if present.
2. A non-default `@Action(output=...)` value on a SEI method MUST result into `wsdl:output` attribute in the corresponding `wsdl:operation`.
3. A non-default `@Action(@FaultAction=...)` value on a SEI method MUST result into `wsdl:fault` attribute in the corresponding `wsdl:operation`. The `wsdl:fault` element MUST correspond to the exception specified by `className` annotated element value.
4. If `wsdl:input[@wsam:Action]` cannot be mapped from the above steps, then `wsam:Action` is generated using the metadata defaulting algorithm as if `wsdl:input[@name]` is not present in WSDL.
5. If `wsdl:output[@wsam:Action]` cannot be mapped from the above steps, then `wsam:Action` is generated using the metadata defaulting algorithm as if `wsdl:output[@name]` is not present in WSDL.
6. If `wsdl:fault[@wsam:Action]` cannot be mapped from the above steps, then `wsam:Action` is generated using the metadata defaulting algorithm as if `wsdl:fault[@name]` is the corresponding exception class name.

For example:

```



```

4.6. Method Parameters and Return Type

A Java method's parameters and return type are mapped to components of either the messages or the global element declarations mapped from the method. Parameters can be mapped to components of the message or global element declaration for either the operation input message, operation output message or both. The mapping depends on the parameter classification. The `javax.jws.WebParam` annotation's `header` element MAY be used to map parameters to SOAP headers. Header parameters MUST be included as `soap:header` elements in the operation's input message. The `javax.jws.WebResult` annotation's `header` element MAY be used to map results to SOAP headers. Header results MUST be included as `soap:header` elements in the operation's output message.

Since JAX-WS uses JAXB for its data binding, JAXB annotations on methods and method parameters MUST be honored. A JAXB annotation on the method is used to specify the binding of a method's return type while an annotation on the parameter specifies the binding of that parameter.

Conformance (use of JAXB annotations): An implementation MUST honor any JAXB annotation that exists on an SEI method or parameter to assure that the proper XML infoset is used when marshalling/unmarshalling the return value or parameters of the method. The set of JAXB annotations that MUST be supported are: `javax.xml.bind.annotation.XmlAttachmentRef`, `javax.xml.bind.annotation.XmlList`, `javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter` and `javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter`.

JAXB doesn't define any namespace by default to types and elements. In the web services, typically these entities that are created for method parameters and return parameters are qualified.

Conformance (Overriding JAXB types empty namespace): JAX-WS tools and runtimes MUST override the default empty namespace for JAXB types and elements to SEI's `targetNamespace`.

4.6.1. Parameter and Return Type Classification

Method parameters and return type are classified as follows:

in

The value is transmitted by copy from a service client to the SEI but is not returned from the service endpoint to the client.

out

The value is returned by copy from an SEI to the client but is not transmitted from the client to the service endpoint implementation.

in/out

The value is transmitted by copy from a service client to the SEI and is returned by copy from the SEI to the client.

A methods return type is always **out**. For method parameters, holder classes are used to determine the classification. `javax.xml.ws.Holder`. A parameter whose type is a parameterized `javax.xml.ws.Holder<T>` class is classified as **in/out** or **out**, all other parameters are classified as **in**.

Conformance (Parameter classification): The `javax.jws.WebParam` annotation (see [mdwebparam]) MAY be used to specify whether a holder parameter is treated as **in/out** or **out**. If not specified, the default MUST be **in/out**.

Conformance (Parameter naming): The `javax.jws.WebParam` annotation (see [mdwebparam]) MAY be used to specify the **name** of the `wsdl:part` or XML Schema element declaration corresponding to a Java parameter. If both the **name** and **partName** elements are used in the `javax.jws.WebParam` annotation then the **partName** MUST be used for the `wsdl:part` name attribute and the **name** element from the annotation will be ignored. If not specified, the default is `arg_N_`, where *N* is replaced with the zero-based argument index. Thus, for instance, the first argument of a method will have a default parameter name of `arg0`, the second one `arg1` and so on.

Conformance (Result naming): The `javax.jws.WebResult` annotation (see [mdwebparam]) MAY be used to specify the **name** of the `wsdl:part` or XML Schema element declaration corresponding to the Java method return type. If both the **name** and **partName** elements are used in the `javax.jws.WebResult` annotations then the **partName** MUST be used for the `wsdl:part` name attribute and the **name** element from the annotation will be ignored. In the absence of customizations, the default name is **return**.

Conformance (Header mapping of parameters and results): The `javax.jws.WebParam` annotation's **header** element MAY be used to map parameters to SOAP headers. Header parameters MUST be included as `soap:header` elements in the operation's input message. The `javax.jws.WebResult` annotation's **header** element MAY be used to map results to SOAP headers. Header results MUST be included as `soap:header` elements in the operation's output message.

4.6.2. Use of JAXB

JAXB 2.1 defines a mapping from Java classes to XML Schema constructs. JAX-WS uses this mapping to generate XML Schema named type and global element declarations that are referred to from within the WSDL **message** constructs generated for each operation.

Three styles of Java to WSDL mapping are supported: document wrapped, document bare and RPC.

The styles differ in what XML Schema constructs are generated for a method. The three styles are described in the following subsections.

The `javax.jws.SOAPBinding` annotation MAY be used to specify at the type level which style to use for all methods it contains or on a per method basis if the `style` is `document`.

4.6.2.1. Document Wrapped

This style is identified by a `javax.jws.SOAPBinding` annotation with the following properties: a `style` of `DOCUMENT`, a `use` of `LITERAL` and a `parameterStyle` of `WRAPPED`.

For the purposes of utilizing the JAXB mapping, each method is converted to two Java bean classes: one for the method input (henceforth called the *request bean*) and one for the method output (henceforth called the *response bean*).

Application's programming model doesn't use these bean classes, so the applications need not package these classes. JAX-WS implementations may generate these classes dynamically as specified in this section.

Conformance (Dynamically generating wrapper beans): A JAX-WS implementation SHOULD not require an application to package request and response bean classes. However, when the bean classes are packaged, they MUST be used.

Conformance (Default wrapper bean names): In the absence of customizations, the wrapper request bean class MUST be named the same as the method and the wrapper response bean class MUST be named the same as the method with a Response suffix. The first letter of each bean name is capitalized to follow Java class naming conventions.

Conformance (Default wrapper bean package): In the absence of customizations, the wrapper beans package MUST be a generated `jaxws` subpackage of the SEI package.

The `javax.xml.ws.RequestWrapper` and `javax.xml.ws.ResponseWrapper` annotations (see `javax.xml.ws.RequestWrapper` and `javax.xml.ws.ResponseWrapper`) MAY be used to customize the name of the generated wrapper bean classes.

Conformance (Wrapper element names): The `javax.xml.ws.RequestWrapper` and `javax.xml.ws.ResponseWrapper` annotations (see `javax.xml.ws.RequestWrapper` and `javax.xml.ws.ResponseWrapper`) MAY be used to specify the qualified name of the elements generated for the wrapper beans.

Conformance (Wrapper bean name clash): Generated bean classes must have unique names within a package and MUST NOT clash with other classes in that package. Clashes during generation MUST be reported as an error and require user intervention via name customization to correct. Note that some platforms do not distinguish filenames based on case so comparisons MUST ignore case.

The name of `wsdl:part` for the wrapper must be named as `parameters` for input messages in the generated WSDL. If a SEI method doesn't have any header parameters or return type, then the `name` of

`wsdl:part` for the wrapper must be named as `parameters` for output messages in the generated WSDL, otherwise it would be named as `result`. The `RequestWrapper` and `ResponseWrapper` annotations MAY be used to customize the `name` of the `wsdl:part` for the wrapper part.

Conformance (Default Wrapper `wsdl:part` names): In the absence of customizations, the name of the `wsdl:part` for the wrapper MUST be named as `parameters` for input messages in the generated WSDL. In the absence of customizations, when there are no header parameters or return type in a SEI method, the `name` of the `wsdl:part` for the wrapper MUST be named as `parameters` for output messages. In all other cases, it MUST be named as `result`.

Conformance (Customizing Wrapper `wsdl:part` names): Non-default `partName` values of the `RequestWrapper` and `ResponseWrapper` annotations, if specified on a SEI method, MUST be used as `wsdl:part name` for the corresponding input and output messages in the generated WSDL.

A request bean is generated containing properties for each `in` and `in/out` non-header parameter. A response bean is generated containing properties for the method return value, each `out` non-header parameter, and `in/out` non-header parameter. Method return values are represented by an `out` property named `return`. The order of the properties in the request bean is the same as the order of parameters in the method signature. The order of the properties in the response bean is the property corresponding to the return value (if present) followed by the properties for the parameters in the same order as the parameters in the method signature.

If a SEI's method parameter or return type is annotated with `@XmlElement`, that annotation is used for the wrapper bean properties. This can be used to map corresponding XML schema element declaration's attributes like `minOccurs`, `maxOccurs`, and `nillable` etc. It is an error to specify `@XmlElement` with a parameter or return type that is mapped to header part. If both `@XmlElement` and `@WebParam`/`@WebResult` are present, then it is an error to specify `@XmlElement`'s `name`, and `namespace` elements different from `@WebParam`/`@WebResult`'s `name` and `targetNamespace` elements respectively.

Conformance (Wrapper property): If a SEI's method parameter or return type is annotated with `@XmlElement`, that annotation MUST be used on the wrapper bean property.

The request and response beans are generated with the appropriate JAXB customizations to result in a global element declaration for each bean class when mapped to XML Schema by JAXB. The corresponding global element declarations MUST NOT have the `nillable` attribute set to a value of `true`. Whereas the element name is derived from the `RequestWrapper` or `ResponseWrapper` annotations, its type is named according to the operation name (for the local part) and the target namespace for the portType that contains the operation (for the namespace name).

Figure [3.3] illustrates this conversion.

```

float getPrice(@WebParam(name="tickerSymbol") String sym);

@XmlRootElement(name="getPrice", targetNamespace="...")
@XmlType(name="getPrice", targetNamespace="...")
@XmlAccessorType(AccessType.FIELD)
public class GetPrice {
    @XmlElement(name="tickerSymbol", targetNamespace="")
    public String tickerSymbol;
}

@XmlRootElement(name="getPriceResponse", targetNamespace="...")
@XmlType(name="getPriceResponse", targetNamespace="...")
@XmlAccessorType(AccessType.FIELD)
public class GetPriceResponse {
    @XmlElement(name="return", targetNamespace="")
    public float _return;
}

```

Figure 3.3: Wrapper mode bean representation of an operation

When the JAXB mapping to XML Schema is utilized this results in global element declarations for the mapped request and response beans with child elements for each method parameter according to the parameter classification:

in

The parameter is mapped to a child element of the global element declaration for the request bean.

out

The parameter or return value is mapped to a child element of the global element declaration for the response bean. In the case of a parameter, the class of the value of the holder class (see section [Parameter and Return Type Classification](#)) is used for the mapping rather than the holder class itself.

in/out

The parameter is mapped to a child element of the global element declarations for the request and response beans. The class of the value of the holder class (see section [Parameter and Return Type Classification](#)) is used for the mapping rather than the holder class itself.

The global element declarations are used as the values of the `wsdl:part` elements `element` attribute, see figure [exinterfacemap].

4.6.2.2. Document Bare

This style is identified by a `javax.jws.SOAPBinding` annotation with the following properties: a `style` of `DOCUMENT`, a `use` of `LITERAL` and a `parameterStyle` of `BARE`.

In order to qualify for use of bare mapping mode a Java method must fulfill all of the following criteria:

1. It must have at most one `in` or `in/out` non-header parameter.
2. If it has a return type other than `void` it must have no `in/out` or `out` non-header parameters.
3. If it has a return type of `void` it must have at most one `in/out` or `out` non-header parameter.

If present, the type of the input parameter is mapped to a named XML Schema type using the mapping defined by JAXB. If the input parameter is a holder class then the class of the value of the holder is used instead.

If present, the type of the output parameter or return value is mapped to a named XML Schema type using the mapping defined by JAXB. If an output parameter is used then the class of the value of the holder class is used.

A global element declaration is generated for the method input and, in the absence of a `WebParam` annotation, its local name is equal to the operation name. A global element declaration is generated for the method output and, in the absence of a `WebParam` or `WebResult` annotation, the local name is equal to the operation name suffixed with `Response`. The type of the two elements depends on whether a type was generated for the corresponding element or not:

Named type generated

The type of the global element is the named type.

No type generated

The type of the element is an anonymous empty type.

The namespace name of the input and output global elements is the value of the `targetNamespace` attribute of the WSDL `definitions` element.

The nillable attribute of the generated global elements MUST have a value of `true` if and only if the corresponding Java types are reference types.

The global element declarations are used as the values of the `wsdl:part` elements `element` attribute, see figure [\[exinterfacemap\]](#).

4.6.2.3. RPC

This style is identified by a `javax.jws.SOAPBinding` annotation with the following properties: a `style` of `RPC`, a `use` of `LITERAL` and a `parameterStyle` of `WRAPPED` [6: Use of `RPC` style requires use of `WRAPPED` parameter style. Deviations from this is an error].

The Java types of each `in`, `out` and `in/out` parameter and the return value are mapped to named XML Schema types using the mapping defined by JAXB. For `out` and `in/out` parameters the class of the value of the holder is used rather than the holder itself.

Each method parameter and the return type is mapped to a message part according to the parameter classification:

in

The parameter is mapped to a part of the input message.

out

The parameter or return value is mapped to a part of the output message.

in/out

The parameter is mapped to a part of the input and output message.

The named types are used as the values of the `wsdl:part` elements `type` attribute, see figure [3.2]. The value of the `name` attribute of each `wsdl:part` element is the name of the corresponding method parameter or return for the method return value.

Due to the limitations described in section 5.3.1 of the WS-I Basic Profile specification (see [8]), null values cannot be used as method arguments or as the return value from a method which uses the `rpc/literal` binding.

Conformance (Null Values in `rpc/literal`): If a null value is passed as an argument to a method, or returned from a method, that uses the `rpc/literal` style, then an implementation **MUST** throw a `WebServiceException`.

4.7. Service Specific Exception

A service specific Java exception is mapped to a `wsdl:fault` element, a `wsdl:message` element with a single child `wsdl:part` element and an XML Schema global element declaration. The `wsdl:fault` element appears as a child of the `wsdl:operation` element that corresponds to the Java method that throws the exception and refers to the `wsdl:message` element. The `wsdl:part` element refers to an XML Schema global element declaration that describes the fault.

Conformance (Exception naming): In the absence of customizations, the name of the global element declaration for a mapped exception **MUST** be the name of the Java exception. The `javax.xml.ws.WebFault` annotation **MAY** be used to customize the local name and namespace name of the element.

Conformance (`wsdl:message` naming): In the absence of customizations, the name of the `wsdl:message` element **MUST** be the name of the Java exception.

The `javax.xml.ws.WebFault` annotation may be used to customize the name of the `wsdl:message` element and also to resolve any conflicts.

Conformance (`wsdl:message` naming using `WebFault`): If an exception has `@WebFault`, then `messageName` **MUST** be the name of the corresponding `wsdl:message` element.

Service specific exceptions are defined as all checked exceptions except `java.rmi.RemoteException` and

its subclasses.

Conformance (`java.lang.RuntimeExceptions` and `java.rmi.RemoteExceptions`): `java.lang.RuntimeException` and `java.rmi.RemoteException` and their subclasses MUST NOT be treated as service specific exceptions and MUST NOT be mapped to WSDL.

JAXB defines the mapping from a Java bean to XML Schema element declarations and type definitions and is used to generate the global element declaration that describes the fault. For exceptions that match the pattern described in section [Fault](#) (i.e. exceptions that have a `getFaultInfo` method and `WebFault` annotation), the *FaultBean* is used as input to JAXB when mapping the exception to XML Schema. For exceptions that do not match the pattern described in section [Fault](#), JAX-WS maps those exceptions to Java beans and then uses those Java beans as input to the JAXB mapping. The following algorithm is used to map non-matching exception classes to the corresponding Java beans for use with JAXB:

1. In the absence of customizations, the name of the bean is the same as the name of the Exception suffixed with Bean.
2. In the absence of customizations, the package of the bean is a generated `jaxws` subpackage of the SEI package. E.g. if the SEI package is `com.example.stockquote` then the package of the bean would be `com.example.stockquote.jaxws`.
3. For each getter in the exception and its superclasses, a property of the same type and name is added to the bean. The `getCause`, `getLocalizedMessage` and `getStackTrace` getters from `java.lang.Throwable` and the `getter from java.lang.Object are excluded from the list of getters to be mapped.`
4. The bean is annotated with a JAXB `@XmlType` annotation. If the exception class has a `@XmlType` annotation, then it is used for the fault bean's `@XmlType` annotation. Otherwise, the fault bean's `@XmlType` annotation is computed with name property set to the name of the exception and the namespace property set to the target namespace of the corresponding portType
 1. Additionally, the `@XmlType` annotation has a `propOrder` property whose value is an array containing the names of all the properties of the exception class that were mapped in the previous bullet point, sorted lexicographically according to the Unicode value of each of their characters (i.e. using the same algorithm that the `int java.lang.String.compareTo(String)` method uses).
5. The bean is annotated with a JAXB `@XmlRootElement` annotation whose `name` property is set, in the absence of customizations, to the name of the exception.

Conformance (Fault bean's @XmlType): If an exception class has a `@XmlType` annotation, then it MUST be used for the fault bean's `@XmlType` annotation.

Conformance (Fault bean name clash): Generated bean classes must have unique names within a package and MUST NOT clash with other classes in that package. Clashes during generation MUST be reported as an error and require user intervention via name customization to correct. Note that some platforms do not distinguish filenames based on case so comparisons MUST ignore case.

Figure [3.4] illustrates this mapping.

```
@WebFault(name="UnknownTickerFault", targetNamespace="...")
public class UnknownTicker extends Exception {
    ...
    public UnknownTicker(String ticker) { ... }
    public UnknownTicker(String ticker, String message) { ... }
    public UnknownTicker(String ticker, String message, Throwable cause)
        { ... }
    public String getTicker() { ... }
}

@XmlRootElement(name="UnknownTickerFault" targetNamespace="...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name="UnknownTicker", namespace="...",
    propOrder={"message", "ticker"})
public class UnknownTickerBean {
    ...
    public UnknownTickerBean() { ... }
    public String getTicker() { ... }
    public void setTicker(String ticker) { ... }
    public String getMessage() { ... }
    public void setMessage(String message) { ... }
}
```

Figure 3.4: Mapping of an exception to a bean for use with JAXB.

Application's programming model doesn't use these bean classes, so the applications need not package these classes. JAX-WS implementations may generate these classes dynamically as specified in this section.

Conformance (Dynamically generating exception beans): JAX-WS implementations SHOULD not require an application to package exception bean classes. However, when the exception bean classes are packaged, they MUST be used.

4.8. Bindings

In WSDL 1.1, an abstract port type can be bound to multiple protocols.

Conformance (Binding selection): An implementation MUST generate a WSDL binding according to the rules of the binding denoted by the `BindingType` annotation (see [javax.xml.ws.BindingType](#)), if present, otherwise the default is the SOAP 1.1/HTTP binding (see [SOAP Binding](#)).

Each protocol binding extends a common extensible skeleton structure and there is one instance of each such structure for each protocol binding. An example of a port type and associated binding skeleton structure is shown in figure [3.5].

```

<portType name="StockQuoteProvider">
  <operation name="getPrice" parameterOrder="tickerSymbol">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
    <fault message="tns:unknowntickerException"/>
  </operation>
</portType>

<binding name="StockQuoteProviderBinding">
  <!-- binding specific extensions possible here -->
  <operation name="getPrice">
    <!-- binding specific extensions possible here -->
    <input message="tns:getPrice">
      <!-- binding specific extensions possible here -->
    </input>
    <output message="tns:getPriceResponse">
      <!-- binding specific extensions possible here -->
    </output>
    <fault message="tns:unknowntickerException">
      <!-- binding specific extensions possible here -->
    </fault>
  </operation>
</binding>

```

Figure 3.5: WSDL portType and associated binding

The common skeleton structure is mapped from Java as described in the following subsections.

4.8.1. Interface

A Java SEI is mapped to a `wsdl:binding` element and zero or more `wsdl:port` extensibility elements.

The `wsdl:binding` element acts as a container for other WSDL elements that together form the WSDL description of the binding to a protocol of the corresponding `wsdl:portType`. The value of the `name` attribute of the `wsdl:binding` is not significant, by convention it contains the qualified name of the corresponding `wsdl:portType` suffixed with Binding.

The `wsdl:port` extensibility elements define the binding specific endpoint address for a given port, see section [Service and Ports](#).

4.8.2. Method and Parameters

Each method in a Java SEI is mapped to a `wsdl:operation` child element of the corresponding `wsdl:binding`. The value of the `name` attribute of the `wsdl:operation` element is the same as the corresponding `wsdl:operation` element in the bound `wsdl:portType`. The `wsdl:operation` element has `wsdl:input`, `wsdl:output`, and `wsdl:fault` child elements if they are present in the corresponding

`wsdl:operation` child element of the `wsdl:portType` being bound.

4.9. Generics

In JAX-WS when starting from Java and if generics are used in the document wrapped case, implementations are required to use type erasure (see JLS section 4.6 for definition of Type Erasure) when generating the request / response wrapper beans and exception beans except in the case of `Collection`. Type erasure is a mapping from parameterized types or type variables to types that are never parameterized types or type variables. Erasure basically gets rid of all the generic type information from the runtime representation. In the case of `Collection` instead of applying erasure on the `Collection` itself, erasure would be applied to the type of `Collection` i.e it would be `Collection<erasure(T)>`. The following code snippets shows the result of erasure on a wrapper bean that is generated when using generics:

```
public <T extends Shape> T setColor(T shape, Color color) {  
    shape.setColor(color);  
    return shape;  
}
```

The generated wrapper bean would be

```

@XmlRootElement(name = "setColor", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "setColor", namespace = "...")
public class SetColor {

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

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

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

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

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

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

```

The following code snippets shows the resulting wrapper bean when using Collections:

```

public List<Shape> echoShapeList(List<Shape> list) {
    return list;
}

```

The generated wrapper bean would be

```

@XmlRootElement(name = "echoShapeList", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "echoShapeList", namespace = "...")
public class EchoShapeList {

    @XmlElement(name = "arg0", namespace = "")
    private List<Shape> arg0;

    public List<Shape> getArg0() {
        return this.arg0;
    }

    public void setArg0(List<Shape> arg0) {
        this.arg0 = arg0;
    }
}

```

```

public <T> T echoTList(List<T> list) {
    if (list.size() == 0)
        return null;
    return list.iterator().next();
}

```

The generated wrapper bean would be

```

@XmlRootElement(name = "echoTList", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "echoTList", namespace = "...")
public class EchoTList {

    @XmlElement(name = "arg0", namespace = "")
    private List<Object> arg0;

    public List<Object> getArg0() {
        return this.arg0;
    }

    public void setArg0(List<Object> arg0) {
        this.arg0 = arg0;
    }
}

```

```

public List<? extends Shape> setArea(List<? extends Shape> list) {
    Iterator iterator = list.iterator();
    while(iterator.hasNext()) {
        iterator.next().setArea(...);
    }
    return list;
}

```

The generated wrapper bean would be

```

@XmlRootElement(name = "setArea", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "setArea", namespace = "...")
public class SetArea {

    @XmlElement(name = "arg0", namespace = "")
    private List<Shape> arg0;

    public List<Shape> getArg0() {
        return this.arg0;
    }

    public void setArg0(List<Shape> arg0) {
        this.arg0 = arg0;
    }
}

```

4.10. SOAP HTTP Binding

This section describes the additional WSDL binding elements generated when mapping Java to WSDL 1.1 using the SOAP HTTP binding.

Conformance (SOAP binding support): Implementations **MUST** be able to generate SOAP HTTP bindings when mapping Java to WSDL 1.1.

Figure [3.6] shows an example of a SOAP HTTP binding.


```

<binding name="StockQuoteProviderBinding">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
  <operation name="getPrice">
    <soap:operation style="document|rpc"/>
    <input message="tns:getPrice">
      <soap:body use="literal"/>
    </input>
    <output message="tns:getPriceResponse">
      <soap:body use="literal"/>
    </output>
    <fault message="tns:unknowntickerException">
      <soap:fault use="literal"/>
    </fault>
  </operation>
</binding>

```

Figure 3.6: WSDL SOAP HTTP binding

4.10.1. Interface

A Java SEI is mapped to a `soap:binding` child element of the corresponding `wSDL:binding` element plus a `soap:address` child element of any corresponding `wSDL:port` element (see section [j2w11svcports]).

The value of the `transport` attribute of the `soap:binding` is `http://schemas.xmlsoap.org/soap/http`. The value of the `style` attribute of the `soap:binding` is either `document` or `rpc`.

Conformance (SOAP binding style required): Implementations MUST include a `style` attribute on a generated `soap:binding`.

4.10.2. Method and Parameters

Each method in a Java SEI is mapped to a `soap:operation` child element of the corresponding `wSDL:operation`. The value of the `style` attribute of the `soap:operation` is `document` or `rpc`. If not specified, the value defaults to the value of the `style` attribute of the `soap:binding`. WS-I Basic Profile[8] requires that all operations within a given SOAP HTTP binding instance have the same binding style.

The parameters of a Java method are mapped to `soap:body` or `soap:header` child elements of the `wSDL:input` and `wSDL:output` elements for each `wSDL:operation` binding element. The value of the `use` attribute of the `soap:body` is `literal`. Figure [3.7] shows an example using document style, figure [3.8] shows the same example using rpc style.

```

<types>
  <schema targetNamespace="...">

```

```

<xsd:element name="getPrice" type="tns:getPriceType"/>
<xsd:complexType name="getPriceType">
  <xsd:sequence>
    <xsd:element name="tickerSymbol" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="getPriceResponse"
  type="tns:getPriceResponseType"/>
<xsd:complexType name="getPriceResponseType">
  <xsd:sequence>
    <xsd:element name="return" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>
</schema>
</types>

<message name="getPrice">
  <part name="getPrice"
    element="tns:getPrice"/>
</message>

<message name="getPriceResponse">
  <part name="getPriceResponse" element="tns:getPriceResponse"/>
</message>

<portType name="StockQuoteProvider">
  <operation name="getPrice" parameterOrder="tickerSymbol">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
  </operation>
</portType>

<binding name="StockQuoteProviderBinding">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  <operation name="getPrice" parameterOrder="tickerSymbol">
    <soap:operation/>
    <input message="tns:getPrice">
      <soap:body use="literal"/>
    </input>
    <output message="tns:getPriceResponse">
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

```

Figure 3.7: WSDL definition using document style

```

<types>
  <schema targetNamespace="...">
    <xsd:element name="getPrice" type="tns:getPriceType"/>
    <xsd:complexType name="getPriceType">
      <xsd:sequence>
        <xsd:element form="unqualified" name="tickerSymbol"
          type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="getPriceResponse"
      type="tns:getPriceResponseType"/>
    <xsd:complexType name="getPriceResponseType">
      <xsd:sequence>
        <xsd:element form="unqualified" name="return"
          type="xsd:float"/>
      </xsd:sequence>
    </xsd:complexType>
  </schema>
</types>

<message name="getPrice">
  <part name="tickerSymbol" type="xsd:string"/>
</message>

<message name="getPriceResponse">
  <part name="result" type="xsd:float"/>
</message>

<portType name="StockQuoteProvider">
  <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
  </operation>
</portType>

<binding name="StockQuoteProviderBinding">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  <operation name="getPrice">
    <soap:operation/>
    <input message="tns:getPrice">
      <soap:body use="literal"/>
    </input>
    <output message="tns:getPriceResponse">

```

```

        <soap:body use="literal"/>
    </output>
</operation>
</binding>

```

Figure 3.8: WSDL definition using rpc style

4.11. Service and Ports

A Java service implementation class is mapped to a single `wsdl:service` element that is a child of a `wsdl:definitions` element for the appropriate target namespace. The latter is mapped from the value of the `targetNamespace` element of the `WebService` annotation, if non-empty value, otherwise from the package of the Java service implementation class according to the rules in section [Package](#).

In mapping a `@WebService`-annotated class (see [Class](#)) to a `wsdl:service`, the `serviceName` element of the `WebService` annotation are used to derive the service name. The value of the `name` attribute of the `wsdl:service` element is computed according to the JSR-181 [\[16\]](#) specification. It is given by the `serviceName` element of the `WebService` annotation, if present with a non-default value, otherwise the name of the implementation class with the `Servicesuffix` appended to it.

Conformance (Service creation): Implementations MUST be able to map classes annotated with the `javax.jws.WebService` annotation to WSDL `wsdl:service` elements.

A WSDL 1.1 service is a collection of related `wsdl:port` elements. A `wsdl:port` element describes a port type bound to a particular protocol (a `wsdl:binding`) that is available at particular endpoint address.

Each desired port is represented by a `wsdl:port` child element of the single `wsdl:service` element mapped from the Java package. JAX-WS 2.0 allows specifying one port of one binding type for each service defined by the application. Implementations MAY support additional ports, as long as their names do not conflict with the standard one.

[Port selection] The `portName` element of the `WebService` annotation, if present, MUST be used to derive the port name to use in WSDL. In the absence of a `portName` element, an implementation MUST use the value of the `name` element of the `WebService` annotation, if present, suffixed with `Port`. Otherwise, an implementation MUST use the simple name of the class annotated with `WebService` suffixed with `Port`.

Conformance (Port binding): The WSDL port defined for a service MUST refer to a binding of the type indicated by the `BindingType` annotation on the service implementation class (see [Bindings](#)).

Binding specific child extension elements of the `wsdl:port` element define the endpoint address for a port. E.g. see the `soap:address` element described in section [Interface](#).

If the endpoint enables Addressing, that can be indicated in the generated WSDL as per the Addressing 1.0 - Metadata[\[27\]](#).

[Use of Addressing] Endpoint's use of addressing, if any, MUST be indicated in the `wsdl:binding` or

`wsdl:port` sections of the WSDL 1.1 as per WS-Addressing 1.0 - Metadata[27].

Example 1: Possible Policy assertion for `@Addressing` in the generated WSDL:

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

Example 2: Possible Policy assertion for `@Addressing(required=true)` in the generated WSDL:

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

Example 3: Possible Policy assertion for

```
<wsam:Addressing wsp:Optional="true">
  <wsp:Policy>
    <wsam:NonAnonymousResponses/>
  </wsp:Policy>
</wsam:Addressing>
```

Chapter 5. Client APIs

This chapter describes the standard APIs provided for client side use of JAX-WS. These APIs allow a client to create proxies for remote service endpoints and dynamically construct operation invocations.

Conformance requirements in this chapter use the term *implementation* to refer to a client side JAX-WS runtime system.

5.1. javax.xml.ws.Service

Service is an abstraction that represents a WSDL service. A WSDL **service** is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at a particular endpoint address.

Service instances are created as described in section [Service Usage](#). **Service** instances provide facilities to:

- Create an instance of a proxy via one of the **getPort** methods. See section [Proxies](#) for information on proxies.
- Create a **Dispatch** instance via the **createDispatch** method. See section [javax.xml.ws.Dispatch](#) for information on the **Dispatch** interface.
- Create a new port via the **addPort** method. Such ports only include binding and endpoint information and are thus only suitable for creating **Dispatch** instances since these do not require WSDL port type information.
- Configure per-service, per-port, and per-protocol message handlers using a handler resolver (see section [Handler Resolver](#)).
- Configure the **java.util.concurrent.Executor** to be used for asynchronous invocations (see section [Executor](#)).

Conformance (Service completeness): A **Service** implementation MUST be capable of creating proxies, **Dispatch** instances, and new ports.

All the service methods except the static **create** methods and the constructors delegate to **javax.xml.ws.spi.ServiceDelegate**, see section [javax.xml.ws.spi.ServiceDelegate](#).

5.1.1. Service Usage

5.1.1.1. Dynamic case

In the dynamic case, when nothing is generated, a Java SE service client uses **Service.create** to create **Service** instances, the following code illustrates this process.

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service s = Service.create(wsdlLocation, serviceName);
```

The following **create** methods may be used:

create(URL wsdlLocation, QName serviceName) Returns a service object for the specified WSDL document and service name.

create(QName serviceName) Returns a service object for a service with the given name. No WSDL document is attached to the service.

create(URL wsdlLocation, QName serviceName, WebServiceFeature ... features) Returns a service object for the specified WSDL document and service name. The created service needs to be configured with the web service features.

create(QName serviceName, WebServiceFeature ... features) Returns a service object for a service with the given name. No WSDL document is attached to the service. The created service needs to be configured with the web service features.

Conformance (Service Creation Failure): If a **create** method fails to create a service object, it **MUST** throw **WebServiceException**. The cause of that exception **SHOULD** be set to an exception that provides more information on the cause of the error (e.g. an **IOException**).

Conformance (Service creation using features): The created service object **MUST** honor the web service features. If a JAX-WS implementation doesn't understand any passed-in feature, it **MUST** throw **WebServiceException**.

5.1.1.2. Static case

When starting from a WSDL document, a concrete service implementation class **MUST** be generated as defined in section [Service and Port](#). The generated implementation class **MUST** have all the public constructors as shown in the example below.

When using the constructors, the default WSDL location and service name are implicitly taken from the **WebServiceClient** annotation that decorates the generated class.

The following code snippet shows the generated constructors:

```
// Generated Service Class
@WebServiceClient(name="StockQuoteService",
    targetNamespace="http://example.com/stocks",
    wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service {
    public StockQuoteService() {
        super(new URL("http://example.com/stocks.wsdl"),
            new QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(WebServiceFeature ... features) {
        super(new URL("http://example.com/stocks.wsdl"),
            new QName("http://example.com/stocks", "StockQuoteService"),
            features);
    }

    public StockQuoteService(URL wsdlLocation) {
        super(wsdlLocation,
            new QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(URL wsdlLocation, WebServiceFeature ... features) {
        super(wsdlLocation,
            new QName("http://example.com/stocks", "StockQuoteService"),
            features);
    }

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

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

    ...
}
```

5.1.2. Provider and Service Delegate

Internally, the `Service` class delegates all of its functionality to a `ServiceDelegate` object, which is part of the SPI used to allow pluggability of implementations.

For this to work, every `Service` object internally MUST hold a reference to a `javax.xml.ws.spi.ServiceDelegate` object (see [javax.xml.ws.spi.ServiceDelegate](#)) to which it delegates

every non-static method call. The field used to hold the reference MUST be private.

The delegate is set when a new `Service` instance is created, which must necessarily happen when the protected, two-argument constructor defined on the `Service` class is called. The constructor MUST obtain a `Provider` instance (see [Creating Endpoint Objects](#)) and call its `createServiceDelegate` method, passing the two arguments received from its caller and the class object for the instance being created (i.e. `this.getClass()`).

In order to ensure that the delegate is properly constructed, the static `create` method defined on the `Service` class MUST call the protected constructor to create a new service instance, passing the same arguments that it received from the application.

The following code snippet shows an implementation of the `Service` API that satisfies the requirements above:

```
public class Service {

    private ServiceDelegate delegate;

    protected Service(java.net.URL wsdlDocumentLocation,
                      QName serviceName) {
        delegate = Provider.provider()
                      .createServiceDelegate(wsdlDocumentLocation
                                             serviceName,
                                             this.getClass());
    }

    public static Service create(java.net.URL wsdlDocumentLocation,
                                QName serviceName) {
        return new Service(wsdlDocumentLocation, serviceName);
    }

    // begin delegated methods

    public <T> T getPort(Class<T> serviceEndpointInterface) {
        return delegate.getPort(serviceEndpointInterface);
    }

    ...
}
```

5.1.3. Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-WS runtime system. Chapter [Handler Framework](#) describes the handler framework in detail. A `Service` instance provides access to a `HandlerResolver` via

a pair of `getHandlerResolver/setHandlerResolver` methods that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a `Service` instance is used to create a proxy or a `Dispatch` instance then the handler resolver currently registered with the service is used to create the required handler chain. Subsequent changes to the handler resolver configured for a `Service` instance do not affect the handlers on previously created proxies, or `Dispatch` instances.

5.1.4. Executor

`Service` instances can be configured with a `java.util.concurrent.Executor`. The executor will then be used to invoke any asynchronous callbacks requested by the application. The `setExecutor` and `getExecutor` methods of `Service` can be used to modify and retrieve the executor configured for a service.

Conformance (Use of Executor): If an executor object is successfully configured for use by a `Service` via the `setExecutor` method, then subsequent asynchronous callbacks MUST be delivered using the specified executor. Calls that were outstanding at the time the `setExecutor` method was called MAY use the previously set executor, if any.

Conformance (Default Executor): Lacking an application-specified executor, an implementation MUST use its own executor, a `java.util.concurrent.ThreadPoolExecutor` or analogous mechanism, to deliver callbacks. An implementation MUST NOT use application-provided threads to deliver callbacks, e.g. by "borrowing" them when the application invokes a remote operation.

5.2. javax.xml.ws.BindingProvider

The `BindingProvider` interface represents a component that provides a protocol binding for use by clients, it is implemented by proxies and is extended by the `Dispatch` interface. Figure [4.1] illustrates the class relationships.

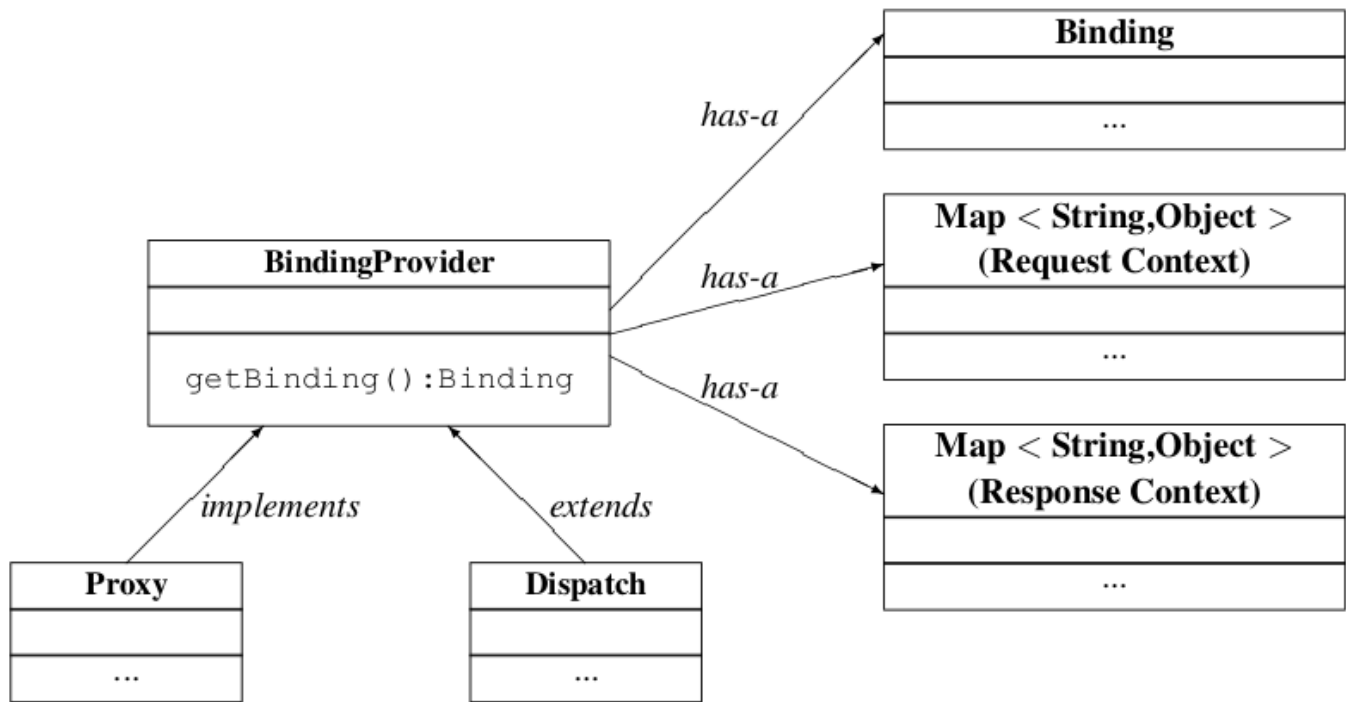


Figure 4.1: Binding Provider Class Relationships

A web service client can get an `javax.xml.ws.EndpointReference` from a `BindingProvider` instance that will reference the target endpoint.

Conformance (`javax.xml.ws.BindingProvider.getEndpointReference`): An implementation **MUST** be able to return an `javax.xml.ws.EndpointReference` for the target endpoint if a SOAP binding is being used. If the `BindingProvider` instance has a binding that is either SOAP 1.1/HTTP or SOAP 1.2/HTTP, then a `W3CEndpointReference` **MUST** be returned. If the binding is XML/HTTP an `java.lang.UnsupportedOperationException` **MUST** be thrown.

Conformance (`BindingProvider's W3CEndpointReference`): The returned `W3CEndpointReference` **MUST** contain `wsam:ServiceName` and `wsam:ServiceName[@EndpointName]` as per Addressing 1.0 - Metadata[27]. The `wsam:InterfaceName` **MAY** be present in the `W3CEndpointReference`. If there is an associated WSDL, then the WSDL location **MUST** be referenced using `wsdl:wsdlLocation` in the `W3CEndpointReference's wsa:Metadata`.

The `BindingProvider` interface provides methods to obtain the `Binding` and to manipulate the binding providers context. Further details on `Binding` can be found in section [javax.xml.ws.Binding](#). The following subsection describes the function and use of context with `BindingProvider` instances.

5.2.1. Configuration

Additional metadata is often required to control information exchanges, this metadata forms the context of an exchange.

A `BindingProvider` instance maintains separate contexts for the request and response phases of a

message exchange with a service:

Request

The contents of the request context are used to initialize the message context (see section [javax.xml.ws.handler.MessageContext](#)) prior to invoking any handlers (see chapter [Handler Framework](#)) for the outbound message. Each property within the request context is copied to the message context with a scope of **HANDLER**.

Response

The contents of the message context are used to initialize the response context after invoking any handlers for an inbound message. The response context is first emptied and then each property in the message context that has a scope of **APPLICATION** is copied to the response context.

Conformance (Message context decoupling): Modifications to the request context while previously invoked operations are in-progress MUST NOT affect the contents of the message context for the previously invoked operations.

The request and response contexts are of type `java.util.Map<String, Object>` and are obtained using the `getRequestContext` and `getResponseContext` methods of `BindingProvider`.

In some cases, data from the context may need to accompany information exchanges. When this is required, protocol bindings or handlers (see chapter [\[handfmwk\]](#)) are responsible for annotating outbound protocol data units and extracting metadata from inbound protocol data units.

Note: *An example of the latter usage: a handler in a SOAP binding might introduce a header into a SOAP request message to carry metadata from the request context and might add metadata to the response context from the contents of a header in a response SOAP message.*

5.2.1.1. Standard Properties

[\[Table 4.1\]](#) lists a set of standard properties that may be set on a `BindingProvider` instance and shows which properties are optional for implementations to support.

Table 4.1: Standard `BindingProvider` properties.

Name	Type	Mandatory	Description
javax.xml.ws.service.endpoint			
.address	String	Y	The address of the service endpoint as a protocol specific URI. The URI scheme must match the protocol binding in use.
javax.xml.ws.security.auth			

.username	String	Y	Username for HTTP basic authentication.
.password	String	Y	Password for HTTP basic authentication.
javax.xml.ws.session			
.maintain	Boolean	Y	Used by a client to indicate whether it is prepared to participate in a service endpoint initiated session. The default value is false .
javax.xml.ws.soap.http.soapaction			
.use	Boolean	N	Controls whether the SOAPAction HTTP header is used in SOAP/HTTP requests. Default value is false.
.uri	String	N	The value of the SOAPAction HTTP header if the <code>javax.xml.ws.soap-http.soapaction.use</code> property is set to true . Default value is an empty string.

Conformance (Required `BindingProvider` properties): An implementation MUST support all properties shown as mandatory in table [Table 4.1].

Note that properties shown as mandatory are not required to be present in any particular context; however, if present, they must be honored.

Conformance (Optional `BindingProvider` properties): An implementation MAY support the properties shown as optional in table [Table 4.1].

5.2.1.2. Additional Properties

Conformance (Additional context properties): Implementations MAY define additional implementation specific properties not listed in table [Table 4.1]. The `java.*` and `javax.*` namespaces are reserved for use by Java specifications.

Implementation specific properties are discouraged as they limit application portability. Applications and binding handlers can interact using application specific properties.

5.2.2. Asynchronous Operations

BindingProvider instances may provide asynchronous operation capabilities. When used, asynchronous operation invocations are decoupled from the **BindingProvider** instance at invocation time such that the response context is not updated when the operation completes. Instead a separate response context is made available using the **Response** interface, see sections [wsdl11asyncmethod] and [dispasyncresp] for further details on the use of asynchronous methods.

Conformance (Asynchronous response context): The local response context of a **BindingProvider** instance MUST NOT be updated on completion of an asynchronous operation, instead the response context MUST be made available via a **Response** instance.

When using callback-based asynchronous operations, an implementation MUST use the **Executor** set on the service instance that was used to create the proxy or **Dispatch** instance being used. See **Executor** for more information on configuring the **Executor** to be used.

5.2.3. Proxies

Proxies provide access to service endpoint interfaces at runtime without requiring static generation of a stub class. See `java.lang.reflect.Proxy` for more information on dynamic proxies as supported by the JDK.

Proxy instances are not guaranteed to be thread safe. If the instances are accessed by multiple threads, usual synchronization techniques can be used to support multiple threads.

Conformance (Proxy support): An implementation MUST support proxies.

*Conformance (Implementing **BindingProvider**):* An instance of a proxy MUST implement `javax.xml.ws.BindingProvider`.

A proxy is created using the `getPort` methods of a **Service** instance:

T `getPort(Class<T> sei)`

Returns a proxy for the specified SEI, the **Service** instance is responsible for selecting the port (protocol binding and endpoint address).

T `getPort(QName port, Class<T> sei)`

Returns a proxy for the endpoint specified by `port`. Note that the namespace component of `port` is the target namespace of the WSDL definitions document.

T `getPort(Class<T> sei, WebServiceFeature... features)`

Returns a proxy for the specified SEI, the **Service** instance is responsible for selecting the port (protocol binding and and endpoint address). The specified `features` MUST be enabled/disabled and configured as specified.

T `getPort(QName port, Class<T> sei, WebServiceFeature... features)`

Returns a proxy for the endpoint specified by `port`. Note that the namespace component of port is

the target namespace of the WSDL definition document. The specified **features** MUST be enabled/disabled and configured as specified.

T `getPort(EndpointReference epr, Class<T> sei, WebServiceFeature... features)`

Returns a proxy for the endpoint specified by **epr**. The address stored in the **epr** MUST be used during invocations on the endpoint. The **epr** MUST NOT be used as the value of any addressing header such as **wsa:ReplyTo**. The specified **features** MUST be enabled/disabled and configured as specified. Any JAX-WS supported **epr** metadata MUST match the **Service** instance's ServiceName, otherwise a **WebServiceException** MUST be thrown. Any JAX-WS supported **epr** metadata MUST match the PortName for the **sei**, otherwise a **WebServiceException** MUST be thrown. If the **Service** instance has an associated WSDL, its WSDL MUST be used to determine any binding information, any WSDL in a JAX-WS supported **epr** metadata MUST be ignored. If the **Service** instance does not have a WSDL, then any WSDL inlined in the JAX-WS supported metadata of the **epr** MUST be used to determine binding information. If there is not enough metadata in the **Service** instance or in the **epr** metadata to determine a port, then a **WebServiceException** MUST be thrown.

The **serviceEndpointInterface** parameter specifies the interface that will be implemented by the proxy. The service endpoint interface provided by the client needs to conform to the WSDL to Java mapping rules specified in chapter [WSDL 1.1 to Java Mapping](#) (WSDL 1.1). Creation of a proxy can fail if the interface doesn't conform to the mapping or if any WSDL related metadata is missing from the **Service** instance.

Conformance (Service.getPort failure): If creation of a proxy fails, an implementation MUST throw **javax.xml.ws.WebServiceException**. The cause of that exception SHOULD be set to an exception that provides more information on the cause of the error (e.g. an **IOException**).

The use of WS-Addressing requirements can be indicated in a WSDL as per Addressing 1.0 - Metadata[27]. A proxy created using **getPort()** calls is configured with the addressing requirements as specified in the associated WSDL or explicitly passing **javax.xml.ws.soap.AddressingFeature** web service feature.

Conformance (Proxy's Addressing use): A proxy MUST be configured with the use of addressing requirements as indicated in the associated WSDL. But if the proxy is created using **javax.xml.ws.soap.AddressingFeature** web service feature, the feature's addressing requirements MUST take precedence over WSDL's addressing requirements.

An implementation is not required to fully validate the service endpoint interface provided by the client against the corresponding WSDL definitions and may choose to implement any validation it does require in an implementation specific manner (e.g., lazy and eager validation are both acceptable).

5.2.3.1. Example

The following example shows the use of a proxy to invoke a method (**getLastTradePrice**) on a service endpoint interface (**com.example.StockQuoteProvider**). Note that no statically generated stub class is involved.


```

javax.xml.ws.Service service = ...;
com.example.StockQuoteProvider proxy = service.getPort(portName,
    com.example.StockQuoteProvider.class)
javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider)proxy;
Map<String, Object> context = bp.getRequestContext();
context.setProperty("javax.xml.ws.session.maintain", Boolean.TRUE);
proxy.getLastTradePrice("ACME");

```

Lines 1–3 show how the proxy is created. Lines 4–6 perform some configuration of the proxy. Lines 7 invokes a method on the proxy.

5.2.4. Exceptions

All methods of an SEI can throw `javax.xml.ws.WebServiceException` and zero or more service specific exceptions.

Conformance (Remote Exceptions): If an error occurs during a remote operation invocation, an implementation **MUST** throw a service specific exception if possible. If the error cannot be mapped to a service specific exception, an implementation **MUST** throw a `ProtocolException` or one of its subclasses, as appropriate for the binding in use. See section [Protocol Specific Exception Handling](#) for more details.

Conformance (Exceptions During Handler Processing): Exceptions thrown during handler processing on the client **MUST** be passed on to the application. If the exception in question is a subclass of `WebServiceException` then an implementation **MUST** rethrow it as-is, without any additional wrapping, otherwise it **MUST** throw a `WebServiceException` whose cause is set to the exception that was thrown during handler processing.

Conformance (Other Exceptions): For all other errors, i.e. all those that don't occur as part of a remote invocation or handler processing, an implementation **MUST** throw a `WebServiceException` whose cause is the original local exception that was thrown, if any.

For instance, an error in the configuration of a proxy instance may result in a `WebServiceException` whose cause is a `java.lang.IllegalArgumentException` thrown by some implementation code.

5.3. javax.xml.ws.Dispatch

XML Web Services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The `Dispatch` interface provides support for this mode of interaction.

Conformance (Dispatch support): Implementations **MUST** support the `javax.xml.ws.Dispatch` interface.

`Dispatch` supports two usage modes, identified by the constants `javax.xml.ws.Service.Mode.MESSAGE` and `javax.xml.ws.Service.Mode.PAYLOAD` respectively:

Message

In this mode, client applications work directly with protocol-specific message structures. E.g., when used with a SOAP protocol binding, a client application would work directly with a SOAP message.

Message Payload

In this mode, client applications work with the payload of messages rather than the messages themselves. E.g., when used with a SOAP protocol binding, a client application would work with the contents of the SOAP **Body** rather than the SOAP message as a whole.

`Dispatch` is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. `Dispatch` is a generic class that supports input and output of messages or message payloads of any type. Implementations are required to support the following types of object:

`javax.xml.transform.Source`

Use of `Source` objects allows clients to use XML generating and consuming APIs directly. `Source` objects may be used with any protocol binding in either message or message payload mode. When used with the HTTP binding (see chapter [HTTP Binding](#)) in payload mode, the HTTP request and response entity bodies must contain XML directly or a MIME wrapper with an XML root part. A `null` value for `Source` is allowed to make it possible to invoke an HTTP GET method in the HTTP Binding case. A `WebServiceException` MUST be thrown when a `Dispatch<Source>` is invoked and the Service returns a MIME message. When used in message mode, if the message is not an XML message a `WebServiceException` MUST be thrown.

JAXB Objects

Use of JAXB allows clients to use JAXB objects generated from an XML Schema to create and manipulate XML representations and to use these objects with JAX-WS without requiring an intermediate XML serialization. JAXB objects may be used with any protocol binding in either message or message payload mode. When used with the HTTP binding (see chapter [HTTP Binding](#)) in payload mode, the HTTP request and response entity bodies must contain XML directly or a MIME wrapper with an XML root part. When used in message mode, if the message is not an XML message a `WebServiceException` MUST be thrown.

`javax.xml.soap.SOAPMessage`

Use of `SOAPMessage` objects allows clients to work with SOAP messages using the convenience features provided by the `java.xml.soap` package. `SOAPMessage` objects may only be used with `Dispatch` instances that use the SOAP binding (see chapter [SOAP Binding](#)) in message mode.

`javax.activation.DataSource`

Use of `DataSource` objects allows clients to work with MIME-typed messages. `DataSource` objects may only be used with `Dispatch` instances that use the HTTP binding (see chapter [HTTP Binding](#)) in message mode.

A JAX-WS implementation MUST honor all `WebServiceFeatures` (section [javax.xml.ws.WebServiceFeature](#)) for `Dispatch` based applications.

5.3.1. Configuration

`Dispatch` instances are obtained using the `createDispatch` factory methods of a `Service` instance. The `mode` parameter of `createDispatch` controls whether the new `Dispatch` instance is message or message payload oriented. The `type` parameter controls the type of object used for messages or message payloads. `Dispatch` instances are not thread safe.

`Dispatch` instances are not required to be dynamically configurable for different protocol bindings; the WSDL binding from which the `Dispatch` instance is generated contains static information including the protocol binding and service endpoint address. However, a `Dispatch` instance may support configuration of certain aspects of its operation and provides methods (inherited from `BindingProvider`) to dynamically query and change the values of properties in its request and response contexts – see section [Standard Properties](#) for a list of standard properties.

5.3.2. Operation Invocation

A `Dispatch` instance supports three invocation modes:

Synchronous request response (`invoke` methods)

The method blocks until the remote operation completes and the results are returned.

Asynchronous request response (`invokeAsync` methods)

The method returns immediately, any results are provided either through a callback or via a polling object.

One-way (`invokeOneWay` methods)

The method is logically non-blocking, subject to the capabilities of the underlying protocol, no results are returned.

Calling `invoke` on the different `Dispatch` types defined above with a `null` value means an empty message will be sent where allowed by the binding, message mode and the MEP. So for example when using -

- SOAP 1.1 / HTTP binding in payload mode using `null` will send a soap message with an empty body.
- SOAP 1.1 / HTTP binding in message mode `null` being passed to `invoke` is an error condition and will result in a `WebServiceException`.
- XML / HTTP binding both in payload and in message mode `null` being passed to `invoke` with the HTTP POST and PUT operations is an error condition and will result in a `WebServiceException`.

Conformance (Failed `Dispatch.invoke`): When an operation is invoked using an `invoke` method, an implementation MUST throw a `WebServiceException` if there is any error in the configuration of the `Dispatch` instance or a `ProtocolException` if an error occurs during the remote operation invocation.

Conformance (Failed `Dispatch.invokeAsync`): When an operation is invoked using an `invokeAsync` method, an implementation MUST throw a `WebServiceException` if there is any error in the configuration of the `Dispatch` instance. Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation.

Conformance (Failed `Dispatch.invokeOneWay`): When an operation is invoked using an `invokeOneWay` method, an implementation MUST throw a `WebServiceException` if there is any error in the configuration of the `Dispatch` instance or if an error is detected [7: The invocation is logically non-blocking so detection of errors during operation invocation is dependent on the underlying protocol in use. For SOAP/HTTP it is possible that certain HTTP level errors may be detected.] during the remote operation invocation.

See section [HTTP](#) for additional SOAP/HTTP requirements.

5.3.3. Asynchronous Response

`Dispatch` supports two forms of asynchronous invocation:

Polling

The `invokeAsync` method returns a `Response` (see below) that may be polled using the methods inherited from `Future<T>` to determine when the operation has completed and to retrieve the results.

Callback

The client supplies an `AsyncHandler` (see below) and the runtime calls the `handleResponse` method when the results of the operation are available. The `invokeAsync` method returns a wildcard `Future` (`Future<?>`) that may be polled to determine when the operation has completed. The object returned from `Future<?>.get()` has no standard type. Client code should not attempt to cast the object to any particular type as this will result in non-portable behavior.

In both cases, errors that occur during the invocation are reported via an exception when the client attempts to retrieve the results of the operation.

Conformance (Reporting asynchronous errors): If the operation invocation fails, an implementation MUST throw a `java.util.concurrent.ExecutionException` from the `Response.get` method.

The cause of an `ExecutionException` is the original exception raised. In the case of a `Response` instance this can only be a `WebServiceException` or one of its subclasses.

The following interfaces are used to obtain the results of an operation invocation:

`javax.xml.ws.Response`

A generic interface that is used to group the results of an invocation with the response context. `Response` extends `java.util.concurrent.Future<T>` to provide asynchronous result polling capabilities.

javax.xml.ws.AsyncHandler

A generic interface that clients implement to receive results in an asynchronous callback. It defines a single `handleResponse` method that has a `Response` object as its argument.

5.3.4. Using JAXB

`Service` provides a `createDispatch` factory method for creating `Dispatch` instances that contain an embedded `JAXBContext`. The `context` parameter contains the `JAXBContext` instance that the created `Dispatch` instance will use to marshall and unmarshall messages or message payloads.

Conformance (Marshalling failure): If an error occurs when using the supplied `JAXBContext` to marshall a request or unmarshall a response, an implementation MUST throw a `WebServiceException` whose `cause` is set to the original `JAXBException`.

5.3.5. Examples

The following examples demonstrate use of `Dispatch` methods in the synchronous, asynchronous polling, and asynchronous callback modes. For ease of reading, error handling has been omitted.

5.3.5.1. Synchronous, Payload-Oriented

```
Source reqMsg = ...;
Service service = ...;
Dispatch<Source> disp = service.createDispatch(portName,
    Source.class, PAYLOAD);
Source resMsg = disp.invoke(reqMsg);
```

5.3.5.2. Synchronous, Message-Oriented

```
SOAPMessage soapReqMsg = ...;
Service service = ...;
Dispatch<SOAPMessage> disp = service.createDispatch(portName,
    SOAPMessage.class, MESSAGE);
SOAPMessage soapResMsg = disp.invoke(soapReqMsg);
```

5.3.5.3. Synchronous, Payload-Oriented With JAXB Objects

```

JAXBContext jc = JAXBContext.newInstance("primer.po");
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po = (PurchaseOrder)u.unmarshal(
    new FileInputStream( "po.xml" ) );
Service service = ...;
Dispatch<Object> disp = service.createDispatch(portName, jc, PAYLOAD);
OrderConfirmation conf = (OrderConfirmation)disp.invoke(po);

```

In the above example `PurchaseOrder` and `OrderConfirmation` are interfaces pre-generated by JAXB from the schema document `primer.po`.

5.3.5.4. Asynchronous, Polling, Message-Oriented

```

SOAPMessage soapReqMsg = ...;
Service service = ...;
Dispatch<SOAPMessage> disp = service.createDispatch(portName,
    SOAPMessage.class, MESSAGE);
Response<SOAPMessage> res = disp.invokeAsync(soapReqMsg);
while (!res.isDone()) {
    // do something while we wait
}
SOAPMessage soapResMsg = res.get();

```

5.3.5.5. Asynchronous, Callback, Payload-Oriented

```

class MyHandler implements AsyncHandler<Source> {
    ...
    public void handleResponse(Response<Source> res) {
        Source resMsg = res.get();
        // do something with the results
    }
}

Source reqMsg = ...;
Service service = ...;
Dispatch<Source> disp = service.createDispatch(portName,
    Source.class, PAYLOAD);
MyHandler handler = new MyHandler();
disp.invokeAsync(reqMsg, handler);

```

5.4. Catalog Facility

JAX-WS mandates support for a standard catalog facility to be used when resolving any Web service

document that is part of the description of a Web service, specifically WSDL and XML Schema documents.

The facility in question is the OASIS XML Catalogs 1.1 specification [34]. It defines an entity catalog that handles the following two cases:

- Mapping an external entity's public identifier and/or system identifier to a URI reference.
- Mapping the URI reference of a resource to another URI reference.

Using the entity catalog, an application can package one or more description and/or schema documents in jar files, avoiding costly remote accesses, or remap remote URIs to other, possibly local ones. Since the catalog is an XML document, a deployer can easily alter it to suit the local environment, unbeknownst to the application code.

The catalog is assembled by taking into account all accessible resources whose name is `META-INF/jax-ws-catalog.xml`. Each resource MUST be a valid entity catalog according to the XML Catalogs 1.1 specification. When running on the Java SE platform, the current context class loader MUST be used to retrieve all the resources with the specified name. Relative URIs inside a catalog file are relative to the location of the catalog that contains them.

Conformance (Use of the Catalog): In the process of resolving a URI that points to a WSDL document or any document reachable from it, a JAX-WS implementation MUST perform a URI resolution for it, as prescribed by the XML Catalogs 1.1 specification, using the catalog defined above as its entity catalog.

In particular, every JAX-WS API argument or annotation element whose semantics is that of a WSDL location URI MUST undergo URI resolution using the catalog facility described in this section.

Although defined in the client API chapter for reasons of ease of exposure, use of the catalog is in no way restricted to client uses of WSDL location URIs. In particular, resolutions of URIs to WSDL and schema documents that arise during the publishing of the contract for an endpoint (see [Determining the Contract for an Endpoint](#)) are subject to the requirements in this section, resulting in catalog-based URI resolutions.

5.5. javax.xml.ws.EndpointReference

A `javax.xml.ws.EndpointReference` is an abstraction that represents an invocable web service endpoint. Client applications can use an `EndpointReference` to get a port for an SEI although doing so prevents them from getting/setting the `Executor` or `HandlerResolver` which would normally be done on a `Service` instance. The `EndpointReference` class delegates to the `javax.xml.ws.spi.Provider` to perform the `getPort` operation. The following method can be used to get a proxy for a Port.

`getPort(Class<T> serviceEndpointInterface, WebServiceFeature... features)`

Gets a proxy for the `serviceEndpointInterface` that can be used to invoke operations on the endpoint referred to by the `EndpointReference` instance. The specified `features` MUST be enabled/disabled and configured as specified. The returned proxy MUST use the `EndpointReference` instance to determine the endpoint address and any reference parameters to be sent on endpoint invocations. The

`EndpointReference` instance MUST NOT be used directly as the value of an WS-Addressing header such as `wsa:ReplyTo`. For this method to successfully return a proxy, WSDL metadata MUST be available and the `EndpointReference` instance MUST contain an implementation understood `ServiceName` in its metadata.

Chapter 6. Service APIs

This chapter describes requirements on JAX-WS service implementations and standard APIs provided for their use.

6.1. javax.xml.ws.Provider

JAX-WS services typically implement a native Java service endpoint interface (SEI), perhaps mapped from a WSDL port type, either directly or via the use of annotations. Section [Interface](#) describes the requirements that a Java interface must meet to qualify as a JAX-WS SEI. Section [Port Type](#) describes the mapping from a WSDL port type to an equivalent Java SEI.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The [Provider](#) interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML message level.

Conformance (Provider support required): An implementation MUST support [Provider<Source>](#) in payload mode with all the predefined bindings. It MUST also support [Provider<SOAPMessage>](#) in message mode in conjunction with the predefined SOAP bindings and [Provider<javax.activation.DataSource>](#) in message mode in conjunction with the predefined HTTP binding.

Conformance (Provider default constructor): A [Provider](#) based service endpoint implementation MUST provide a public default constructor.

An empty Source payload can be used in payload mode to send a response with no payload. An empty source can be constructed using zero-argument default constructors of DOMSource, SAXSource, and StreamSource.

A typed [Provider](#) interface is one in which the type parameter has been bound to a concrete class, e.g. [Provider<Source>](#) or [Provider<SOAPMessage>](#), as opposed to being left unbound, as in [Provider<T>](#).

Conformance (Provider implementation): A [Provider](#) based service endpoint implementation MUST implement a typed [Provider](#) interface.

Conformance (WebServiceProvider annotation): A [Provider](#) based service endpoint implementation MUST carry a [WebServiceProvider](#) annotation (see [javax.xml.ws.WebServiceProvider](#)).

[Provider](#) is a low level generic API that requires services to work with messages or message payloads and hence requires an intimate knowledge of the desired message or payload structure. The generic nature of [Provider](#) allows use with a variety of message object types.

A JAX-WS implementation MUST honor all [WebServiceFeatures](#) (section [javax.xml.ws.WebServiceFeature](#)) for [Provider](#) based applications.

6.1.1. Invocation

A **Provider** based service instance's **invoke** method is called for each message received for the service.

When an invoke method returns null, it is considered that no response needs to be sent by service.

6.1.1.1. Exceptions

The service runtime is required to catch exceptions thrown by a Provider instance. A **Provider** instance may make use of the protocol specific exception handling mechanism as described in section [Protocol Specific Exception Handling](#). The protocol binding is responsible for converting the exception into a protocol specific fault representation and then invoking the handler chain and dispatching the fault message as appropriate.

6.1.2. Configuration

The **ServiceMode** annotation is used to configure the messaging mode of a **Provider** instance. Use of **@ServiceMode(value=MESSAGE)** indicates that the provider instance wishes to receive and send entire protocol messages (e.g. a SOAP message when using the SOAP binding); absence of the annotation or use of **@ServiceMode(value=PAYLOAD)** indicates that the provider instance wishes to receive and send message payloads only (e.g. the contents of a SOAP Body element when using the SOAP binding).

Provider instances MAY use the **WebServiceContext** facility (see [javax.xml.ws.WebServiceContext](#)) to access the message context and other information about the request currently being served.

The JAX-WS runtime makes certain properties available to a **Provider** instance that can be used to determine its configuration. These properties are passed to the **Provider** instance each time it is invoked using the **MessageContext** instance accessible from the **WebServiceContext**.

6.1.3. Examples

For brevity, error handling is omitted in the following examples.

6.1.3.1. Simple echo service, reply message is the same as the input message

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

    public SOAPMessage invoke(SOAPMessage request) {
        return request;
    }
}
```

6.1.3.2. Simple static reply, reply message contains a fixed acknowledgment element

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class MyService implements Provider<Source> {
    public MyService() {
    }

    public Source invoke(Source request) {
        Source requestPayload = request.getPayload();
        ...
        String replyElement = "<n:ack xmlns:n='...' />";
        StreamSource reply = new StreamSource(new StringReader(replyElement));
        return reply;
    }
}
```

6.1.3.3. Using JAXB to read the input message and set the reply

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class MyService implements Provider<Source> {
    public MyService() {
    }

    public Source invoke(Source request) {
        JAXBContent jc = JAXBContext.newInstance(...);
        Unmarshaller u = jc.createUnmarshaller();
        Object requestObj = u.unmarshall(request);
        ...
        Acknowledgement reply = new Acknowledgement(...);
        return new JAXBSource(jc, reply);
    }
}
```

6.2. javax.xml.ws.Endpoint

The `Endpoint` class can be used to create and publish Web service endpoints.

An endpoint consists of an object that acts as the Web service implementation (called here *implementor*) plus some configuration information, e.g. a `Binding`. Implementor and binding are set when the endpoint is created and cannot be modified later. Their values can be retrieved using the `getImplementor` and `getBinding` methods respectively. Other configuration information may be set at any time after the creation of an `Endpoint` but before its publication.

6.2.1. Endpoint Usage

Endpoints can be created using the following static methods on `Endpoint`:

`create(Object implementor)`

Creates and returns an `Endpoint` for the specified implementor. If the implementor specifies a binding using the `javax.xml.ws.BindingType` annotation it MUST be used else a default binding of SOAP 1.1 / HTTP binding MUST be used.

`create(Object implementor, WebServiceFeature ... features)`

Same as the above `create()` method. The created `Endpoint` is configured with the web service features. These features override the corresponding features that are specified in WSDL, if present.

`create(String bindingID, Object implementor)`

Creates and returns an `Endpoint` for the specified binding and implementor. If the bindingID is `null` and no binding information is specified via the `javax.xml.ws.BindingType` annotation then a default SOAP 1.1 / HTTP binding MUST be used.

`create(String bindingID, Object implementor, WebServiceFeature ... features)`

Same as the above `create()` method. The created `Endpoint` is configured with the web service features. These features override the corresponding features that are specified in WSDL, if present.

`publish(String address, Object implementor)`

Creates and publishes an `Endpoint` for the given implementor. The binding is chosen by default based on the URL scheme of the provided address (which must be a URL). If a suitable binding is found, the endpoint is created then published as if the `Endpoint.publish(String address)` method had been called. The created `Endpoint` is then returned as the value of the method.

`publish(String address, Object implementor, WebServiceFeature ... features)`

Same as the above `publish()` method. The created `Endpoint` is configured with the web service features. These features override the corresponding features that are specified in WSDL, if present.

These methods MUST delegate the creation of `Endpoint` to the `javax.xml.ws.spi.Provider` SPI class (see [javax.xml.ws.spi.Provider](#)) by calling the `createEndpoint` and `createAndPublishEndpoint` methods respectively.

An implementor object MUST be either an instance of a class annotated with the `@WebService` annotation according to the rules in chapter [Java to WSDL 1.1 Mapping](#) or an instance of a class annotated with the `WebServiceProvider` annotation and implementing the `Provider` interface (see [javax.xml.ws.Provider](#)).

The `publish(String, Object)` method is provided as a shortcut for the common operation of creating and publishing an `Endpoint`. The following code provides an example of its use:

```
// assume Test is an endpoint implementation class annotated with @WebService
Test test = new Test();
Endpoint e = Endpoint.publish("http://localhost:8080/test", test);
```

Conformance (Endpoint publish(String address, Object implementor) Method): The effect of invoking the `publish` method on an `Endpoint` MUST be the same as first invoking the `create` method with the binding ID appropriate to the URL scheme used by the address, then invoking the `publish(String address)` method on the resulting `endpoint`.

Conformance (Default Endpoint Binding): In the absence of a specified binding, if the URL scheme for the address argument of the `Endpoint.publish` method is `http` or `https` then an implementation MUST use the SOAP 1.1/HTTP binding (see chapter [SOAP Binding](#)) as the binding for the newly created endpoint.

Conformance (Other Bindings): An implementation MAY support using the `Endpoint.publish` method with addresses whose URL scheme is neither `http` nor `https`.

The success of the `Endpoint.publish` method is conditional to the presence of the appropriate permission as described in section [Publishing Permission](#).

Endpoint implementors MAY use the `WebServiceContext` facility (see [javax.xml.ws.WebServiceContext](#)) to access the message context and other information about the request currently being served. Injection of the `WebServiceContext`, if requested, MUST happen the first time the endpoint is published. After any injections have been performed and before any requests are dispatched to the implementor, the implementor method which carries a `javax.annotation.PostConstruct` annotation, if present, MUST be invoked. Such a method MUST satisfy the requirements for lifecycle methods in JSR-250 [\[35\]](#).

6.2.2. Publishing

An `Endpoint` is in one of three states: not published (the default), published or stopped. Published endpoints are active and capable of receiving incoming requests and dispatching them to their implementor. Non published endpoints are inactive. Stopped endpoint were in the published until some time ago, then got stopped. Stopped endpoints cannot be published again. Publication of an `Endpoint` can be achieved by invoking one of the following methods:

`publish(String address)`

Publishes the endpoint at the specified address (a URL). The address MUST use a URL scheme compatible with the endpoint's binding.

`publish(Object serverContext)`

Publishes the endpoint using the specified server context. The server context MUST contain address information for the resulting endpoint and it MUST be compatible with the endpoint's binding.

Conformance (Publishing over HTTP): If the `Binding` for an `Endpoint` is a SOAP (see [SOAP Binding](#)) or HTTP (see [HTTP Binding](#)) binding, then an implementation MUST support publishing the `Endpoint` to a

URL whose scheme is either `http''` or `https''`.

The WSDL contract for an endpoint is created dynamically based on the annotations on the implementor class, the `Binding` in use and the set of metadata documents specified on the endpoint (see [Endpoint Metadata](#)).

Conformance (WSDL Publishing): An `Endpoint` that uses the SOAP 1.1/HTTP binding (see [SOAP Binding](#)) MUST make its contract available as a WSDL 1.1 document at the publishing address suffixed with `?WSDL''` or `?wsdl''`.

An `Endpoint` that uses any other binding defined in this specification in conjunction with the HTTP transport SHOULD make its contract available using the same convention. It is RECOMMENDED that an implementation provide a way to access the contract for an endpoint even when the latter is published over a transport other than HTTP.

The success of the two `Endpoint.publish` methods described above is conditional to the presence of the appropriate permission as described in section [Publishing Permission](#).

Applications that wish to modify the configuration information (e.g. the metadata) for an `Endpoint` must make sure the latter is in the not-published state. Although the various setter methods on `Endpoint` must always store their arguments so that they can be retrieved by a later invocation of a getter, the changes they entail may not be reflected on the endpoint until the next time it is published. In other words, the effects of configuration changes on a currently published endpoint are undefined.

The `stop` method can be used to stop publishing an endpoint. A stopped endpoint may not be restarted. It is an error to invoke a `publish` method on a stopped endpoint. After the `stop` method returns, the runtime MUST NOT dispatch any further invocations to the endpoint's implementor.

An `Endpoint` will be typically invoked to serve concurrent requests, so its implementor should be written so as to support multiple threads. The `synchronized` keyword may be used as usual to control access to critical sections of code. For finer control over the threads used to dispatch incoming requests, an application can directly set the executor to be used, as described in section [Executor](#).

6.2.2.1. Example

The following example shows the use of the `publish(Object)` method using a hypothetical HTTP server API that includes the `HttpServer` and `HttpContext` classes.

```
// assume Test is an endpoint implementation class annotated with @WebService
Test test = new Test();
HttpServer server = HttpServer.create(new InetSocketAddress(8080), 10);
server.setExecutor(Executor.newFixedThreadPool(10));
server.start();
HttpContext context = server.createContext("/test");
Endpoint endpoint = Endpoint.create(SOAPBinding.SOAP11HTTP_BINDING, test);
endpoint.publish(context);
```

Note that the specified server context uses its own executor mechanism. At runtime then, any other executor set on the `Endpoint` instance would be ignored by the JAX-WS implementation.

6.2.3. Publishing Permission

For security reasons, administrators may want to restrict the ability of applications to publish Web service endpoints. To this end, JAX-WS 2.0 defines a new permission class, `javax.xml.ws.WebServicePermission`, and one named permission, `publishEndpoint`.

Conformance (Checking `publishEndpoint` Permission): When any of the `publish` methods defined by the `Endpoint` class are invoked, an implementation MUST check whether a `SecurityManager` is installed with the application. If it is, implementations MUST verify that the application has the `WebServicePermission` identified by the target name `publishEndpoint` before proceeding. If the permission is not granted, implementations MUST NOT publish the endpoint and they MUST throw a `java.lang.SecurityException`.

6.2.4. Endpoint Metadata

A set of metadata documents can be associated with an `Endpoint` by means of the `setMetadata(List<Source>)` method. By setting the metadata of an `Endpoint`, an application can bypass the automatic generation of the endpoint's contract and specify the desired contract directly. This way it is possible, e.g., to make sure that the WSDL or XML Schema document that is published contains information that cannot be represented using built-in Java annotations (see [Annotations](#)).

Conformance (Required Metadata Types): An implementation MUST support WSDL 1.1 and XML Schema 1.0 documents as metadata.

Conformance (Unknown Metadata): An implementation MUST ignore metadata documents whose type it does not recognize.

When specifying a list of documents as metadata, an application may need to establish references between them. For instance, a WSDL document may import one or more XML Schema documents. In order to do so, the application MUST use the `systemId` property of the `javax.xml.transform.Source` class by setting its value to an absolute URI that uniquely identifies it among all supplied metadata documents, then using the given URI in the appropriate construct (e.g. `wSDL:import` or `xsd:import`).

6.2.5. Determining the Contract for an Endpoint

This section details how the annotations on the endpoint implementation class and the metadata for an endpoint instance are used at publishing time to create a contract for the endpoint.

Both the `WebService` and `WebServiceProvider` annotations define a `wSDLLocation` annotation element which can be used to point to the desired WSDL document for the endpoint. If such an annotation element is present on the endpoint implementation class and has a value other than the default one (i.e. it is not the empty string), then a JAX-WS implementation MUST use the document referred to from the `wSDLLocation` annotation element to determine the contract, according to the rules in section [Use of](#)

[@WebService\(wsdlLocation\)](#) and [Metadata](#).

In addition to the case in which the [Endpoint](#) API is explicitly used, the requirements in this section are also applicable to the publishing of an endpoint via declarative means, e.g. in a servlet container. In this case, there may not be an equivalent for the notion of metadata as described in [Endpoint Metadata](#). In such an occurrence, the rules in this section MUST be applied using an empty set of metadata documents as the metadata for the endpoint.

In the context of the Java EE Platform, JSR-109 [17] defines deployment descriptor elements that may be used to override the value of the [wsdlLocation](#) annotation element. Please refer to that specification for more details.

As we specify additional rules to be used in determining the contract for an endpoint, we distinguish two cases: that of a SEI-based endpoint (i.e. an endpoint that is annotated with a [WebService](#) annotation) and that of a Provider-based endpoint.

6.2.5.1. SEI-based Endpoints

For publishing to succeed, a SEI-based endpoint MUST have an associated contract.

If the [wsdlLocation](#) annotation element is the empty string, then a JAX-WS implementation must obey the following rules, depending on the binding used by the endpoint:

SOAP 1.1/HTTP Binding

A JAX-WS implementation MUST generate a WSDL description for the endpoint based on the rules in section [Use of @WebService\(wsdlLocation\) and Metadata](#) below.

SOAP 1.2/HTTP Binding

A JAX-WS implementation MUST NOT generate a WSDL description for the endpoint.

HTTP Binding

A JAX-WS implementation MUST NOT generate a WSDL description for the endpoint.

Any Implementation-Specific Binding

A JAX-WS implementation MAY generate a WSDL description for the endpoint.

This requirements guarantee that future versions of this specification may mandate support for additional WSDL binding in conjunction with the predefined binding identifiers without negatively affecting existing applications.

A generated contract MUST follow the rules in chapter [Java to WSDL 1.1 Mapping](#) and those in the JAXB specification [10].

6.2.5.2. Provider-based Endpoints

Provider-based endpoints SHOULD have a non-empty [wsdlLocation](#) pointing to a valid WSDL description of the endpoint.

If the `wSDLLocation` annotation element is the empty string, then a JAX-WS implementation MUST NOT generate a WSDL description for the endpoint.

6.2.5.3. Use of `@WebService(wSDLLocation)` and Metadata

A WSDL document contains two different kinds of information: abstract information (i.e. portTypes and any schema-related information) which affects the format of the messages and the data being exchanged, and binding-related one (i.e. bindings and ports) which affects the choice of protocol and transport as well as the on-the-wire format of the messages. Annotations (see [Annotations](#)) are provided to capture the former aspects but not the latter. (The `@SOAPBinding` annotation is a bit of a hybrid, because it captures the signature-related aspects of the `soap:binding` binding extension in WSDL 1.1.)

At runtime, annotations must be followed for all the abstract aspects of an interaction, but binding information has to come from somewhere else. Although the choice of binding is made at the time an endpoint is created, this specification does not attempt to capture all possible binding properties in its APIs, since the extensibility of WSDL would make it a futile exercise. Rather, when an endpoint is published, a description for it, if present, is consulted to determine binding information, using the `wSDL:service` and `wSDL:port` qualified names as a key.

In terms of priority, the description specified using the `wSDLLocation` annotation element, if present, comes first, and the metadata documents are secondary. In the absence of a non-empty, non-default `wSDLLocation` annotation element, the metadata documents are consulted to identify as many description components as possible that can be reused when producing the contract for the endpoint.

There are some restrictions on the packaging of the description and any associated metadata documents. The goal of these restrictions is to make it possible to publish an endpoint without forcing a JAX-WS implementation to retrieve, store and patch multiple documents from potentially remote sites.

The value of the `wSDLLocation` annotation element on an endpoint implementation class, if any, MUST be a relative URL. The document it points to MUST be packaged with the application. Moreover, it MUST follow the requirements in section [Application-specified Service](#) below ("Application-specified Service").

In the Java SE platform, relative URLs are treated as resources. When running on the Java EE platform, the dispositions in the JSR-109 specification apply.

For ease of identification, let's call this document the "root description document", to distinguish it from any WSDL documents it might import.

At publishing time, a JAX-WS implementation MUST patch the endpoint address in the root description document to match the actual address the endpoint is deployed at.

In order to state the requirements for patching the locations of any `wSDL:import`-ed or `xsd:import`-ed documents, let's define a document as being *local* if and only if

1. it is the root description document, or
2. it is reachable from a local document via an import statement whose location is either a relative URL or an absolute URL for which there is a corresponding metadata document (i.e. a `Source` object which is a member of the list of metadata documents and whose `systemId` property is equal to the URL in question).

A JAX-WS implementation **MUST** patch the location attributes of all `wSDL:import` and `xsd:import` statement in local documents that point to local documents. An implementation **MUST NOT** patch any other location attributes.

Please note that, although the catalog facility (see [Catalog Facility](#)) is used to resolve any absolute URLs encountered while processing the root description document or any documents transitively reachable from it via `wSDL:import` and `xsd:import` statements, those absolute URLs will not be rewritten when the importing document is published, since documents resolved via the catalog are not considered local, even if the catalog maps them to resources packaged with the application.

In what follows, for better readability, the term `'metadata document'` should be interpreted as also covering the description document pointed to by the `'wsdlLocation'` annotation element (if any), while keeping in mind the processing rules in the preceding paragraphs.

As a guideline, the generated contract must reuse as much as possible the set of metadata documents provided by the application. In order to simplify an implementor's task, this specification requires that only a small number of well-defined scenarios in which the application provides metadata documents be supported.

Implementations **MAY** support other use cases, but they **MUST** follow the general rule that any application-provided metadata element takes priority over an implementation-generated one, with the exception of the overriding of a port address.

For instance, if the application-provided metadata contains a definition for portType *foo* that in no case should the JAX-WS implementation create its own *foo* portType to replace the one provided by the application in the final contract for the endpoint.

The exception to using a metadata document as supplied by the application without any modifications is the address of the `wSDL:port` for the endpoint, which **MUST** be overridden so as to match the address specified as an argument to the `publish` method or the one implicit in a server context.

When publishing the main WSDL document for an endpoint, an implementation **MUST** ensure that all references between documents are correct and resolvable. This may require remapping the metadata documents to URLs different from those set as their `systemId` property. The renaming **MUST** be consistent, in that the `imports` and `includes` relationships existing between documents when the metadata was supplied to the endpoint **MUST** be respected at publishing time. Moreover, the same metadata document **SHOULD NOT** be published at multiple, different URLs.

When resolving URI references to other documents when processing metadata documents or any of the documents they may transitively reference, a JAX-WS implementation **MUST** use the catalog facility

defined in section [Catalog Facility](#), except when there is a metadata document whose system id matches the URI in question. In other words, metadata documents have priority over catalog-based mappings.

The scenarios which are required to be supported are the following:

6.2.5.4. Application-specified Service

One of the metadata documents, say **D**, contains a definition for a WSDL service whose qualified name, say **S**, matches that specified by the endpoint being published. In this case, a JAX-WS implementation **MUST** use **D** as the service description. No further generation of contract-related artifacts may occur.

Table 5.1: Standard **Endpoint** properties.

Name	Type	Description
javax.xml.ws.wsdl		
.service	QName	Specifies the qualified name of the service.
.port	QName	Specifies the qualified name of the port.

The implementation **MUST** also override the port address in **D** and the **location** and **schemaLocation** attributes as detailed in the preceding paragraphs. It is an error if more than one metadata document contains a definition for the sought-after service **S**.

6.2.5.5. Application-specified PortType

No metadata document contains a definition for the sought-after service **S**, but a metadata document, say **D**, contains a definition for the WSDL portType whose qualified name, say **P**, matches that specified by the endpoint being published. In this case, a JAX-WS implementation **MUST** create a new description for **S**, including an appropriate WSDL binding element referencing portType **P**. The metadata document **D** **MUST** be imported/included so that the published contract uses the definition of **P** provided by **D**. No schema generation occurs, as **P** is assumed to embed or import schema definitions for all the types/elements it requires. Like in the previous case, the implementation **MUST** override any **location** and **schemaLocation** attributes. It is an error if more than one metadata document contains a definition for the sought-after portType **P**.

6.2.5.6. Application-specified Schema or No Metadata

No metadata document contains a definition for the sought-after service **S** and portType **P**. In this case, a JAX-WS implementation **MUST** generate a complete WSDL for **S**. When it comes to generating a schema for a certain target namespace, say **T**, the implementation **MUST** reuse the schema for **T** among the available metadata documents, if any. Like in the preceding case, the implementation **MUST** override any **schemaLocation** attributes. It is an error if more than one schema documents specified as metadata for the endpoint attempt to define components in a namespace **T** used by the endpoint.

Note: The three scenarios described above cover several applicative use cases. The first one represents an application that has full control over all aspects of the contract. The JAX-WS runtime just uses what the application provided, with a minimum of adjustments to ensure consistency. The second one corresponds to an application that defines all abstract aspects of the WSDL, i.e. `portType(s)` and `schema(s)`, leaving up to the JAX-WS runtime to generate the concrete portions of the contract. Finally, the third case represents an application that uses one or more well-known schema(s), possibly taking advantage of lots of facets/constraints that JAXB cannot capture, and wants to reuse it as-is, leaving all the WSDL-specific aspects of the contract up to the runtime. This use case also covers an application that does not specify any metadata, leaving WSDL and schema generation up to the JAX-WS (and JAXB) implementation.

6.2.6. Endpoint Properties

An `Endpoint` has an associated set of properties that may be read and written using the `getProperties` and `setProperties` methods respectively.

[Table 5.1] lists the set of standard `Endpoint` properties.

When present, the WSDL-related properties override the values specified using the `WebService` and `WebServiceProvider` annotations. This functionality is most useful with provider objects (see section [javax.xml.ws.WebServiceProvider](#)), since the latter are naturally more suited to a more dynamic usage. For instance, an application that publishes a provider endpoint can decide at runtime which web service to impersonate by using a combination of metadata documents and the properties described in this section.

6.2.7. Executor

`Endpoint` instances can be configured with a `java.util.concurrent.Executor`. The executor will then be used to dispatch any incoming requests to the application. The `setExecutor` and `getExecutor` methods of `Endpoint` can be used to modify and retrieve the executor configured for a service.

Conformance (Use of Executor): If an executor object is successfully set on an `Endpoint` via the `setExecutor` method, then an implementation MUST use it to dispatch incoming requests upon publication of the `Endpoint` by means of the `publish(String address)` method. If publishing is carried out using the `publish(Object serverContext)` method, an implementation MAY use the specified executor or another one specific to the server context being used.

Conformance (Default Executor): If an executor has not been set on an `Endpoint`, an implementation MUST use its own executor, a `java.util.concurrent.ThreadPoolExecutor` or analogous mechanism, to dispatch incoming requests.

6.2.8. javax.xml.ws.EndpointReference

The following methods can be used on a published `Endpoint` to retrieve an `javax.xml.ws.EndpointReference` for the `Endpoint` instance.

getEndpointReference(List<Element> referenceParameters)

Creates and returns a `javax.xml.ws.EndpointReference` for a published `Endpoint`. If the binding is SOAP 1.1/HTTP or SOAP 1.2/HTTP, then a `javax.xml.ws.wsaddressing.W3CEndpointReference` MUST be returned. A returned `W3CEndpointReference` MUST also contain the specified `referenceParameters`. An implementation MUST throw a `javax.xml.ws.WebServiceException` if the `Endpoint` instance has not been published. An implementation MUST throw `java.lang.UnsupportedOperationException` if the `Endpoint` instance uses the XML/HTTP binding.

getEndpointReference(Class<T> clazz, List<Element> referenceParameters)

Creates and returns a `javax.xml.ws.EndpointReference` of type `clazz` for a published `Endpoint` instance. If `clazz` is of type `javax.xml.ws.wsaddressing.W3CEndpointReference`, then the returned `W3CEndpointReference` MUST contain the specified `referenceParameters`. An implementation MUST throw a `javax.xml.ws.WebServiceException` if the `Endpoint` instance has not been published. If the Class `clazz` is not a subclass of `EndpointReference` or the `Endpoint` implementation does not support `EndpointReferences` of type `clazz` a `javax.xml.ws.WebServiceException` MUST be thrown. An implementation MUST throw `java.lang.UnsupportedOperationException` if the `Endpoint` instance uses the XML/HTTP binding.

Conformance (Endpoint's `W3CEndpointReference`): The returned `W3CEndpointReference` MUST contain `wsam:ServiceName` and `wsam:ServiceName[@EndpointName]` as per Addressing 1.0 - Metadata[27]. The `wsam:InterfaceName` MAY be present in the `W3CEndpointReference`. If there is an associated WSDL, then the WSDL location MUST be referenced using `wsdl:wsdlLocation` in the `W3CEndpointReference`'s `wsa:Metadata`.

6.3. javax.xml.ws.WebServiceContext

The `javax.xml.ws.WebServiceContext` interface makes it possible for an endpoint implementation object and potentially any other objects that share its execution context to access information pertaining to the request being served.

The result of invoking any methods on the `WebServiceContext` of a component outside the invocation of one of its web service methods is undefined. An implementation SHOULD throw a `java.lang.IllegalStateException` if it detects such a usage.

The `WebServiceContext` is treated as an injectable resource that can be set at the time an endpoint is initialized. The `WebServiceContext` object will then use thread-local information to return the correct information regardless of how many threads are concurrently being used to serve requests addressed to the same endpoint object.

In Java SE, the resource injection denoted by the `WebServiceContext` annotation is REQUIRED to take place only when the annotated class is an endpoint implementation class.

The following code shows a simple endpoint implementation class which requests the injection of its `WebServiceContext`:

```

@WebService
public class Test {
    @Resource
    private WebServiceContext context;
    public String reverse(String inputString) { ... }
}

```

The `javax.annotation.Resource` annotation defined by JSR-250 [35] is used to request injection of the `WebServiceContext`. The following constraints apply to the annotation elements of a `Resource` annotation used to inject a `WebServiceContext`:

- The `type` element MUST be either `java.lang.Object` (the default) or `javax.xml.ws.WebServiceContext`. If the former, then the resource MUST be injected into a field or a method. In this case, the type of the field or the type of the JavaBeans property defined by the method MUST be `javax.xml.ws.WebServiceContext`.
- The `authenticationType`, `shareable` elements, if they appear, MUST have their respective default values.

The above restriction on `type` guarantees that a resource type of `WebServiceContext` is either explicitly stated or can be inferred from the annotated field/method declaration. Moreover, the field/method type must be assignable from the type described by the annotation's `type` element.

When running on the Java SE platform, the `name` and `mappedName` elements are ignored. As a consequence, on Java SE there is no point in declaring a resource of type `WebServiceContext` on the endpoint class itself (instead of one of its fields/methods), since it won't be accessible at runtime via JNDI.

When running on the Java EE 5 platform, resources of type `WebServiceContext` are treated just like all other injectable resources there and are subject to the constraints prescribed by the platform specification [36].

An endpoint implementation can retrieve an `javax.xml.ws.EndpointReference` for the endpoint using `getEndpointReference(List<Element> referenceParameters)`, and `getEndpointReference(Class<T> clazz, List<Element> referenceParameters)` methods. These methods have the same semantics as the `Endpoint.getEndpointReference()` methods specified in the section [javax.xml.ws.EndpointReference](#)

Note: When using method-based injection, it is recommended that the method be declared as non-public, otherwise it will be exposed as a web service operation. Alternatively, the method can be marked with the `@WebMethod(exclude=true)` annotation to ensure it will not be part of the generated portType for the service.

6.3.1. MessageContext

The message context made available to endpoint instances via the `WebServiceContext` acts as a restricted window on to the `MessageContext` of the inbound message following handler execution (see

chapter [Handler Framework](#)). The restrictions are as follows:

- Only properties whose scope is `APPLICATION` are visible using a `MessageContext` obtained from a `WebServiceContext`; the `get` method returns `null` for properties with `HANDLER` scope, the `Set` returned by `keySet` only includes properties with `APPLICATION` scope.
- New properties set in the context are set in the underlying `MessageContext` with `APPLICATION` scope.
- An attempt to set the value of property whose scope is `HANDLER` in the underlying `MessageContext` results in an `IllegalArgumentException` being thrown.
- Only properties whose scope is `APPLICATION` can be removed using the context. An attempt to remove a property whose scope is `HANDLER` in the underlying `MessageContext` results in an `IllegalArgumentException` being thrown.
- The `Map.putAll` method can be used to insert multiple properties at once. Each property is inserted individually, each insert operation being carried out as if enclosed by a try/catch block that traps any `IllegalArgumentException`. Consequently, `putAll` is not atomic: it silently ignores properties whose scope is `HANDLER` and it never throws an `IllegalArgumentException`.

The `MessageContext` is used to store handlers information between request and response phases of a message exchange pattern, restricting access to context properties in this way ensures that endpoint implementations can only access properties intended for their use.

6.4.

javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder

Occasionally it is necessary for one application component to create an `EndpointReference` for another web service endpoint. The `W3CEndpointReferenceBuilder` class provides a standard API for creating `W3CEndpointReference` instances for web service endpoints.

Conformance (Building W3CEndpointReference): `W3CEndpointReferenceBuilder.build()` method MUST construct an `EndpointReference` as per the Addressing 1.0 - Metadata[27].

Chapter 7. Core APIs

This chapter describes the standard core APIs that may be used by both client and server side applications.

7.1. javax.xml.ws.Binding

The `javax.xml.ws.Binding` interface acts as a base interface for JAX-WS protocol bindings. Bindings to specific protocols extend `Binding` and may add methods to configure specific aspects of that protocol binding's operation. Chapter [SOAP Binding](#) describes the JAX-WS SOAP binding; chapter [HTTP Binding](#) describes the JAX-WS XML/HTTP binding.

Applications obtain a `Binding` instance from a `BindingProvider` (a proxy or `Dispatch` instance) or from an `Endpoint` using the `getBinding` method (see sections [javax.xml.ws.BindingProvider](#), [javax.xml.ws.Endpoint](#)).

A concrete binding is identified by a *binding id*, i.e. a URI. This specification defines a number of standard bindings and their corresponding identifiers (see chapters [SOAP Binding](#) and [HTTP Binding](#)). Implementations MAY support additional bindings. In order to minimize conflicts, the identifier for an implementation-specific binding SHOULD use a URI scheme that includes a domain name or equivalent, e.g. the `http` URI scheme. Such identifiers SHOULD include a domain name controlled by the implementation's vendor.

`Binding` provides methods to manipulate the handler chain configured on an instance (see section [Programmatic Configuration](#)).

Conformance (Read-only handler chains): An implementation MAY prevent changes to handler chains configured by some other means (e.g. via a deployment descriptor) by throwing `UnsupportedOperationException` from the `setHandlerChain` method of `Binding`.

7.2. javax.xml.ws.spi.Provider

`Provider` is an abstract service provider interface (SPI) factory class that provides various methods for the creation of `Endpoint` instances and `ServiceDelegate` instances. These methods are designed for use by other JAX-WS API classes, such as `Service` (see [javax.xml.ws.Service](#)) and `Endpoint` (see [javax.xml.ws.Endpoint](#)) and are not intended to be called directly by applications.

The `Provider` SPI allows an application to use a different JAX-WS implementation from the one bundled with the platform without any code changes.

Conformance (Concrete `javax.xml.ws.spi.Provider` required): An implementation MUST provide a concrete class that extends `javax.xml.ws.spi.Provider`. Such a class MUST have a public constructor which takes no arguments.

7.2.1. Configuration

The **Provider** implementation class is determined using the following algorithm. The steps listed below are performed in sequence. At each step, at most one candidate implementation class name will be produced. The implementation will then attempt to load the class with the given class name using the current context class loader or, if missing one, the `java.lang.Class.forName(String)` method. As soon as a step results in an implementation class being successfully loaded, the algorithm terminates.

1. Use the service-provider loading facilities, defined by the `java.util.ServiceLoader` class, to attempt to locate and load an implementation of `javax.xml.ws.spi.Provider` service using the default loading mechanism .
2. Use the configuration file `jaxws.properties`. The file is in standard `java.util.Properties` format and typically located in the conf directory of the Java installation. It contains the fully qualified name of the implementation class with the key `javax.xml.ws.spi.Provider`.
3. If a system property with the name `javax.xml.ws.spi.Provider` is defined, then its value is used as the name of the implementation class.
4. Finally, a platform implementation is used.

7.2.2. Creating Endpoint Objects

Endpoints can be created using the following methods on **Provider**:

createEndpoint(String bindingID, Object implementor) Creates and returns an **Endpoint** for the specified binding and implementor. If the `bindingId` is null and no binding information is specified via the `javax.xml.ws.BindingType` annotation then a default SOAP1.1/HTTP binding MUST be used.

createEndpoint(String bindingID, Object implementor, WebServiceFeature ... features) Same as the above `createEndpoint()` method. The created **Endpoint** is configured with the web service features.

createAndPublishEndpoint(String address, Object implementor) Creates and publishes an **Endpoint** for the given implementor. The binding is chosen by default based on the URL scheme of the provided address (which must be a URL). If a suitable binding is found, the endpoint is created then published as if the `Endpoint.publish(String address)` method had been called. The created **Endpoint** is then returned as the value of the method.

createAndPublishEndpoint(String address, Object implementor, WebServiceFeature ... features) Same as the above `createAndPublishEndpoint()` method. The created **Endpoint** is configured with the web service features.

createEndpoint(String bindingId, Class<?> implementorClass, Invoker invoker, WebServiceFeature ... features) Creates an **Endpoint** for the implementor class and the endpoint invocation is handled by the **Invoker**. If the `bindingId` is null and no binding information is specified via the `javax.xml.ws.BindingType` annotation then a default SOAP1.1/HTTP binding MUST be used. The created **Endpoint** is configured with the web service features.

An implementor object MUST be either:

- an instance of a SEI-based endpoint class, i.e. a class annotated with the `@WebService` annotation according to the rules in chapter [Java to WSDL 1.1 Mapping](#), or
- an instance of a provider class, i.e. a class implementing the `Provider` interface and annotated with the `WebServiceProvider` annotation according to the rules in [javax.xml.ws.Provider](#).

The `createAndPublishEndpoint(String, Object)` method is provided as a shortcut for the common operation of creating and publishing an `Endpoint`. It corresponds to the static `publish` method defined on the `Endpoint` class, see [Endpoint Usage](#).

Conformance (Provider createAndPublishEndpoint Method): The effect of invoking the `createAndPublishEndpoint` method on a `Provider` MUST be the same as first invoking the `createEndpoint` method with the binding ID appropriate to the URL scheme used by the address, then invoking the `publish(String address)` method on the resulting `endpoint`.

7.2.3. Creating ServiceDelegate Objects

`javax.xml.ws.spi.ServiceDelegate` `javax.xml.ws.spi.ServiceDelegate` can be created using the following methods on `Provider`:

`createServiceDelegate(URL wsdlDocumentLocation, QName serviceName, Class <? extends Service> serviceClass)` Creates and returns a `ServiceDelegate` for the specified service. When starting from WSDL the serviceClass will be the generated service class as described in section [Service and Port](#). In the dynamic case where there is no service class generated it will be `javax.xml.ws.Service`. The serviceClass is used by the `ServiceDelegate` to get access to the annotations.

`createServiceDelegate(URL wsdlDocumentLocation, QName serviceName, Class <? extends Service> serviceClass, WebServiceFeature ... features)` Same as the above `createServiceDelegate()` method and it also configures the delegate with all the web service features.

7.2.4. EndpointReferences

The `Provider` class provides the following methods to create `EndpointReference` instances.

`readEndpointReference(javax.xml.transform.Source source)` Unmarshalls and returns a `javax.xml.ws.EndpointReference` from the infoset contained in `source`.

`createW3CEndpointReference` Creates a `W3CEndpointReference` using the specified `String address`, `QName serviceName`, `QName portName`, `List<Element> metadata`, `String wsdlDocumentLocation`, and `List<Element> referenceParameters` parameters.

`createW3CEndpointReference(String address, QName serviceName, QName portName, List<Element> metadata, String wsdlDocumentLocation, List<Element> referenceParameters)` Creates `W3CEndpointReference` using the specified parameters.

`createW3CEndpointReference(String address, QName interfaceName, QName serviceName, QName portName,`

`List<Element> metadata, String wsdlDocumentLocation, List<Element> referenceParameters, List<Element> elements, Map<QName, String> attributes)` Creates `W3CEndpointReference` using the specified parameters. This method adds support for extension elements, extension attributes, and porttype name.

7.2.5. Getting Port Objects

The following method can be used to get a proxy for a Port.

`getPort(EndpointReference epr, Class<T> sei, WebServiceFeature... features)`

Gets a proxy for the `sei` that can be used to invoke operations on the endpoint referred to by the `epr`. The specified `features` MUST be enabled/disabled and configured as specified. The returned proxy MUST use the `epr` to determine the endpoint address and any reference parameters that MUST be sent on endpoint invocations. The `epr` MUST NOT be used directly as the value of an WS-Addressing header such as `wsa:ReplyTo`.

7.3. javax.xml.ws.spi.ServiceDelegate

The `javax.xml.ws.spi.ServiceDelegate` class is an abstract class that implementations MUST provide. This is the class that `javax.xml.ws.Service` `javax.xml.ws.Service` class delegates all methods, except the static `create` methods to. `ServiceDelegate` is defined as an abstract class for future extensibility purpose.

Conformance (Concrete `javax.xml.ws.spi.ServiceDelegate` required): An implementation MUST provide a concrete class that extends `javax.xml.ws.spi.ServiceDelegate`.

7.4. Exceptions

The following standard exceptions are defined by JAX-WS.

`javax.xml.ws.WebServiceException`

A runtime exception that is thrown by methods in JAX-WS APIs when errors occur during local processing.

`javax.xml.ws.ProtocolException`

A base class for exceptions related to a specific protocol binding. Subclasses are used to communicate protocol level fault information to clients and may be used by a service implementation to control the protocol specific fault representation.

`javax.xml.ws.soap.SOAPFaultException`

A subclass of `ProtocolException`, may be used to carry SOAP specific information.

`javax.xml.ws.http.HTTPException`

A subclass of `ProtocolException`, may be used to carry HTTP specific information.

Editors Note 6.1 *A future version of this specification may introduce a new exception class to distinguish*

errors due to client misconfiguration or inappropriate parameters being passed to an API from errors that were generated locally on the sender node as part of the invocation process (e.g. a broken connection or an unresolvable server name). Currently, both kinds of errors are mapped to `WebServiceException`, but the latter kind would be more usefully mapped to its own exception type, much like `ProtocolException` is.

7.4.1. Protocol Specific Exception Handling

Conformance (Protocol specific fault generation): When throwing an exception as the result of a protocol level fault, an implementation **MUST** ensure that the exception is an instance of the appropriate `ProtocolException` subclass. For SOAP the appropriate `ProtocolException` subclass is `SOAPFaultException`, for XML/HTTP is `HTTPException`.

Conformance (Protocol specific fault consumption): When an implementation catches an exception thrown by a service endpoint implementation and the cause of that exception is an instance of the appropriate `ProtocolException` subclass for the protocol in use, an implementation **MUST** reflect the information contained in the `ProtocolException` subclass within the generated protocol level fault.

7.4.1.1. Client Side Example

```
try {
    response = dispatch.invoke(request);
}
catch (SOAPFaultException e) {
    QName soapFaultCode = e.getFault().getFaultCodeAsQName();
    ...
}
```

7.4.1.2. Server Side Example

```
public void endpointOperation() {
    ...
    if (someProblem) {
        SOAPFault fault = soapBinding.getSOAPFactory().createFault(
            faultcode, faultstring, faultactor, detail);
        throw new SOAPFaultException(fault);
    }
    ...
}
```

7.4.1.3. One-way Operations

Conformance (One-way operations): When sending a one-way message, implementations **MUST** throw a `WebServiceException` if any error is detected when sending the message.

7.5. javax.xml.ws.WebServiceFeature

JAX-WS 2.1 introduces the notion of features. A feature is associated with a particular functionality or behavior. Some features may only have meaning when used with certain bindings while other features may be generally useful.

These features can be used while creating service and proxy instances.

JAX-WS 2.1 introduces three standard features

for creating proxy instances

, `AddressingFeature`, `MTOMFeature` and `RespectBindingFeature` as well as the base `WebServiceFeature` class.

There are no standard features for service creation in the current specification. A JAX-WS 2.2 implementation may define its own features but they will be non-portable across all JAX-WS 2.2 implementations.

Each feature is derived from the `javax.xml.ws.WebServiceFeature` class. This allows the web service developer to pass different types of `WebServiceFeatures` to the various JAX-WS APIs that utilize them. Also, each feature should be documented using JavaDocs on the derived classes. Each `WebServiceFeature` MUST have a `public static final String ID` field that is used to uniquely identify the feature.

Conformance (javax.xml.ws.WebServiceFeatures): Each derived type of `javax.xml.ws.WebServiceFeature` MUST contain a `public static final String ID` field that uniquely identifies the feature against all features of all implementations.

Since vendors can specify their own features, care MUST be taken when creating a feature ID so as to not conflict with another vendor's ID.

The `WebServiceFeature` class also has an `enabled` property that is used to store whether a particular feature should be enabled or disabled. Each derived type should provide either a constructor argument and/or a method that will allow the web service developer to set the `enabled` property. The meaning of enabled or disabled is determined by each individual `WebServiceFeature`. It is important that web services developers be able to enable/disable specific features when writing their web applications. For example, a developer may choose to implement WS-Addressing himself while using the Dispatch and Provider APIs and thus he MUST be able to tell JAX-WS to disable addressing.

Conformance (enabled property): Each derived type of `javax.xml.ws.WebServiceFeature` MUST provide a constructor argument and/or method to allow the web service developer to set the value of the `enabled` property. The public default constructor MUST by default set the `enabled` property to `true`. An implementation MUST honor the value of the `enabled` property of any supported `WebServiceFeature`.

7.5.1. javax.xml.ws.soap.AddressingFeature

The `AddressingFeature` is used to control the use of WS-Addressing[26] by JAX-WS. This feature MUST

be supported with the SOAP 1.1/HTTP or SOAP 1.2/HTTP bindings. Using this feature with any other binding is undefined. This feature corresponds to the Addressing annotation described in section [javax.xml.ws.soap.Addressing](#).

Enabling this feature on the server will result in the runtime being capable of consuming and responding to WS-Addressing headers. Enabling this feature on the client will cause the JAX-WS runtime to include WS-Addressing headers in SOAP messages as specified by WS-Addressing[26].

Disabling this feature will prevent a JAX-WS runtime from processing or adding WS-Addressing headers from/to SOAP messages even if the associated WSDL specifies otherwise. This may be necessary if a client or endpoint needs to implement Addressing themselves. For example, a client that desires to use non-anonymous ReplyTo can do so by disabling the `AddressingFeature` and by using `Dispatch<Source>` with `Message` mode.

The `AddressingFeature`'s `required` property can be configured to control whether all incoming messages MUST contain Addressing headers.

The `AddressingFeature`'s `responses` property can be configured to control whether the endpoint requires the use of anonymous, non-anonymous and all responses.

This feature is automatically enabled if the WSDL indicates the use of addressing as per the WS-Addressing 1.0 - Metadata[27]. Developers may choose to prevent this from happening by explicitly disabling the `AddressingFeature`.

7.5.1.1. javax.xml.ws.EndpointReference

The abstract `EndpointReference` class is used by the JAX-WS APIs to reference a particular endpoint in accordance with the W3C Web Services Addressing 1.0[26]. Each concrete instance of an `EndpointReference` MUST contain a `wsa:Address`.

Applications may also use the `EndpointReference` class in method signatures. JAXB 2.1 will bind the `EndpointReference` base class to `xs:anyType`. Applications should instead use concrete implementations of `EndpointReference` such as `javax.xml.ws.W3CEndpointReference` which will provide better binding. JAX-WS implementations are required to support the `W3CEndpointReference` class but they may also provide other `EndpointReference` subclasses that represent different versions of Addressing.

7.5.1.2. javax.xml.ws.W3CEndpointReference

The `W3CEndpointReference` class is a concrete implementation of the `javax.xml.ws.EndpointReference` class and is used to reference endpoints that are compliant with the W3C Web Services Addressing 1.0 - Core[26] recommendation. Applications may use this class to pass `EndpointReference` instances as method parameters or return types. JAXB 2.1 will bind the `W3CEndpointReference` class to the W3C `EndpointReference` XML Schema in the WSDL.

7.5.1.3. javax.xml.ws.soap.MTOMFeature

The `MTOMFeature` is used to specify if MTOM should be used with a web service. This feature should be

used instead of the `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING`, `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING` and the `javax.xml.ws.soap.SOAPBinding.setMTOMEnabled()`. This feature MUST be supported with the SOAP 1.1/HTTP or SOAP 1.2/HTTP bindings. Using this feature with any other bindings is undefined. This feature corresponds to the `MTOM` annotation described in section [javax.xml.ws.soap.MTOM](#).

Enabling this feature on either the server or client will result the JAX-WS runtime using MTOM and for binary data being sent as an attachment.

The `MTOMFeature` has one property `threshold`, that can be configured to serve as a hint for which binary data SHOULD be sent as an attachment. The `threshold` is the size in bytes that binary data SHOULD be in order to be sent as an attachment. The `threshold` MUST not be negative. The default value is `0`.

Conformance (`javax.xml.ws.soap.MTOMFeature`): An implementation MUST support the `javax.xml.ws.soap.MTOMFeature` and its `threshold` property.

7.5.1.4. javax.xml.ws.RespectBindingFeature

The `RespectBindingFeature` is used to control whether a JAX-WS implementation MUST respect/honor the contents of the `wsdl:binding` associated with an endpoint. It has a corresponding `RespectBinding` annotation described in section [javax.xml.ws.RespectBinding](#).

Conformance (`javax.xml.ws.RespectBindingFeature`): When the `javax.xml.ws.RespectBindingFeature` is enabled, a JAX-WS implementation MUST inspect the `wsdl:binding` at runtime to determine result and parameter bindings as well as any `wsdl:extensions` that have the `required=true` attribute. All required `wsdl:extensions` MUST be supported and honored by a JAX-WS implementation unless a specific `wsdl:extension` has been explicitly disabled via a `WebServiceFeature`.

When this feature is enabled, a JAX-WS implementation must support and honor the addressing policy, if specified, in the WSDL. However, such addressing requirements can be explicitly disabled via `AddressingFeature`.

In order to not break backward compatibility with JAX-WS 2.0, the behavior with regards to respecting the `wsdl:binding` when this feature is disabled is undefined.

7.6. javax.xml.ws.spi.http (HTTP SPI)

The classes in this package can be used for a portable deployment of JAX-WS web services in a HTTP container (for example, servlet container). This SPI enables to decouple the JAX-WS deployment and runtime and is not meant for end developers but for container or its extension developers.

The HTTP SPI allows a deployment to use any available web services runtime for HTTP transport. Java EE6 web profile vendors can support JSR-109[17] deployments using the JAX-WS 2.2 runtime in Java SE platform. For example, a Servlet 3.0 extension can be used to do the JSR-109 deployment by reading deployment descriptors and hand-off the request processing to the web services runtime that is in Java

SE platform.

The HTTP SPI consists of the following classes:

`javax.xml.ws.spi.http.HttpContext`

`HttpContext` represents a mapping between the root URI path of a web service to a `Handler` which is invoked to handle requests destined for that path on the associated container.

`javax.xml.ws.spi.http.HttpExchange`

This class encapsulates a HTTP request received and a response to be generated in one exchange.

`javax.xml.ws.spi.http.Handler`

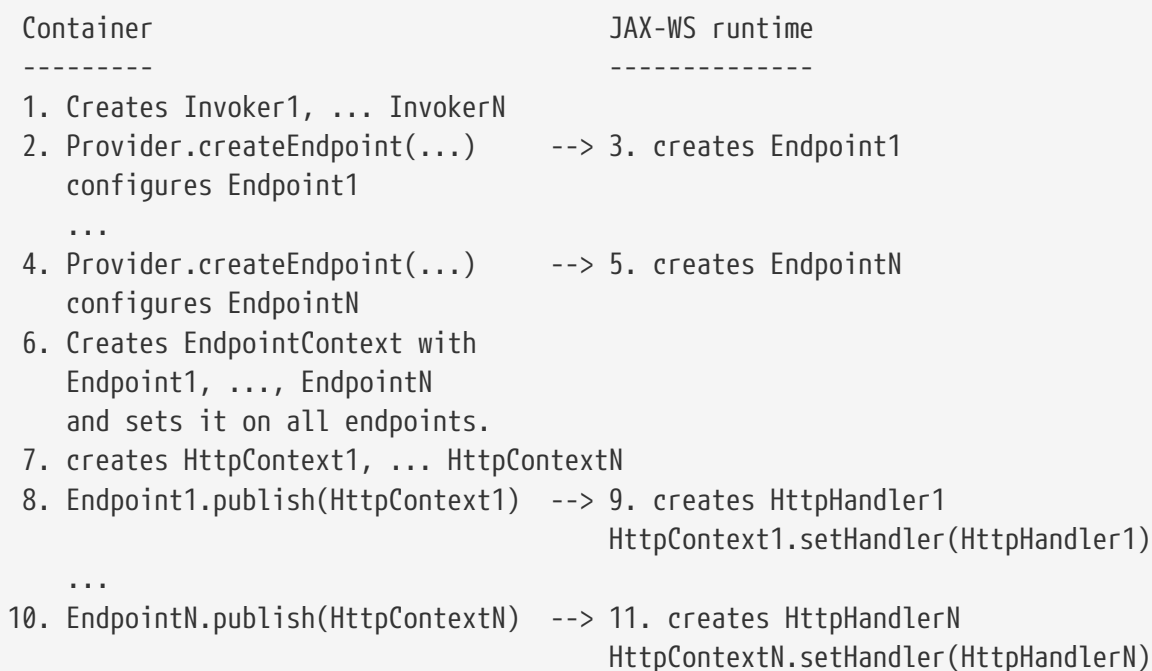
A handler which is invoked to process HTTP exchanges.

`javax.xml.ws.spi.Invoker`

`Invoker` hides the detail of calling into application endpoint implementation.

Typical portable deployment is done as below:

1. Container creates Endpoint objects for an application. The necessary information to create `Endpoint` objects may be got from web service deployment descriptor files.
2. Container creates `HttpContext` objects for the deployment. For example, a `HttpContext` could be created using servlet configuration(for e.g. url-pattern) for a web service in servlet container case.
3. Then publishes all the endpoints using `Endpoint.publish(HttpContext)`. During `publish()`, JAX-WS runtime registers a `Handler` callback to handle incoming requests or `HttpExchange` objects. The `HttpExchange` object encapsulates HTTP request and response.



Typical request processing is done as below(for every request):

Container	JAX-WS runtime
-----	-----
1. Creates a <code>HttpExchange</code>	
2. Gets handler from <code>HttpContext</code>	
3. <code>HttpHandler.handle(HttpExchange)</code>	--> 4. reads request from <code>HttpExchange</code>
	<-- 5. Calls <code>Invoker</code>
6. Invokes the actual instance	
	7. Writes the response to <code>HttpExchange</code>

Typical portable undeployment is done as below:

Container	JAX-WS runtime
-----	-----
1. <code>@preDestroy</code> on instances	
2. <code>Endpoint1.stop()</code>	
...	
3. <code>EndpointN.stop()</code>	

Having a support for this SPI in a JAX-WS implementation in Java SE platform would enable deployments to use the Java SE platform's web services runtime portably.

Conformance (HTTP SPI in SE platform): A JAX-WS 2.2 implementation in Java SE platform MUST support `Endpoint.publish(javax.xml.ws.spi.http.HttpContext)`.

Chapter 8. Annotations

This chapter describes the annotations used by JAX-WS.

For simplicity, when describing an annotation we use the term `property` in lieu of the more correct "annotation elements". Also, for each property we list the default value, which is the default as it appears in the declaration of the annotation type. Often properties have logical defaults which are computed based on contextual information and, for this reason, cannot be captured using the annotation element default facility built into the language. In this case, the text describes what the logical default is and how it is computed.

JAX-WS 2.0 uses annotations extensively. For an annotation to be correct, besides being syntactically correct, e.g. placed on a program element of the appropriate type, it must obey a set of constraints detailed in this specification. For annotations defined by JSR-181, the annotation in question must also obey the constraints in the relevant specification (see [16]).

Conformance (Correctness of annotations): An implementation MUST check at runtime that the annotations pertaining to a method being invoked, either on the client or on the server, as well as any containing program elements (i.e. classes, packages) is in conformance with the specification for that annotation

Conformance (Handling incorrect annotations): If an incorrect or inconsistent annotation is detected:

- In a client setting, an implementation MUST NOT invoke the remote operation being invoked, if any. Instead, it MUST throw a `WebServiceException`, setting its cause to an exception approximating the cause of the error (e.g. an `IllegalArgumentException` or a `ClassNotFoundException`).
- In a server setting, annotation, an implementation MUST NOT dispatch to an endpoint implementation object. Rather, it MUST generate a fault appropriate to the binding in use.

Conformance (Unsupported `WebServiceFeatureAnnotation`): If an unrecognized or unsupported annotation annotated with the `WebServiceFeatureAnnotation` meta-annotation:

- In a client setting, an implementation MUST NOT invoke the remote operation being invoked, if any. Instead, it MUST throw a `WebServiceException`, setting its cause to an exception approximating the cause of the error (e.g. an `IllegalArgumentException` or a `ClassNotFoundException`).
- In a server setting, annotation, an implementation MUST NOT dispatch to an endpoint implementation object. Rather, it MUST generate a fault appropriate to the binding in use.

An implementation may check for correctness in a lazy way, at the time a method is invoked or a request is about to be dispatched to an endpoint, or more aggressively, e.g. when creating a proxy. In a container environment, an implementation may perform any correctness checks at deployment time.

8.1. javax.xml.ws.ServiceMode

The `ServiceMode` annotation is used to specify the mode for a provider class, i.e. whether a provider wants to have access to protocol message payloads (e.g. a SOAP body) or the entire protocol messages (e.g. a SOAP envelope).

Table 7.1: `ServiceMode` properties.

Property	Description	Default
<code>value</code>	The service mode, one of <code>javax.xml.ws.Service.Mode.MESSAGE</code> or <code>javax.xml.ws.Service.Mode.PAYLOAD</code> . <code>MESSAGE</code> means that the whole protocol message will be handed to the provider instance, <code>PAYLOAD</code> that only the payload of the protocol message will be handed to the provider instance.	<code>javax.xml.ws.Service.Mode.PAYLOAD</code>

The `ServiceMode` annotation type is marked `@Inherited`, so the annotation will be inherited from the superclass.

8.2. javax.xml.ws.WebFault

The `WebFault` annotation is used when mapping WSDL faults to Java exceptions, see section [Fault](#). It is used to capture the name of the fault element used when marshalling the JAXB type generated from the global element referenced by the WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults.

Table 7.2: `WebFault` properties.

Property	Description	Default
<code>name</code>	The local name of the element	""
<code>targetNamespace</code>	The namespace name of the element	""
<code>faultBean</code>	The fully qualified name of the fault bean class	""
<code>messageName</code>	The name of the <code>wsdl:message</code>	""

8.3. javax.xml.ws.RequestWrapper

The `RequestWrapper` annotation is applied to the methods of an SEI. It is used to capture the JAXB generated request wrapper bean and the element name and namespace for marshalling /

unmarshalling the bean. The default value of `localName` element is the `operationName` as defined in `WebMethod` annotation and the default value for the `targetNamespace` element is the target namespace of the SEI. When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the `className` element is required in this case.

Table 7.3: `RequestWrapper` properties.

Property	Description	Default
<code>localName</code>	The local name of the element	""
<code>targetNamespace</code>	The namespace name of the element	""
<code>className</code>	The name of the wrapper class	""
<code>partName</code>	The name of the <code>wsdl:part</code>	""

8.4. javax.xml.ws.ResponseWrapper

The `ResponseWrapper` annotation is applied to the methods of an SEI. It is used to capture the JAXB generated response wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of the `localName` element is the `operationName` as defined in the `WebMethod` appended with `'Response'` and the default value of the `'targetNamespace'` element is the target namespace of the SEI. When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the `className` element is required in this case.

Table 7.4: `ResponseWrapper` properties.

Property	Description	Default
<code>localName</code>	The local name of the element	""
<code>targetNamespace</code>	The namespace name of the element	""
<code>className</code>	The name of the wrapper class	""
<code>partName</code>	The name of the <code>wsdl:part</code>	""

8.5. javax.xml.ws.WebServiceClient

The `WebServiceClient` annotation is specified on a generated service class (see [Service and Port](#)). It is used to associate a class with a specific Web service, identify by a URL to a WSDL document and the qualified name of a `wsdl:service` element.

Table 7.5: `ResponseWrapper` properties.

Property	Description	Default
<code>name</code>	The local name of the service	""

<code>targetNamespace</code>	The namespace name of the service	""
<code>wsdlLocation</code>	The URL for the WSDL description of the service	""

When resolving the URI specified as the `wsdlLocation` element or any document it may transitively reference, a JAX-WS implementation **MUST** use the catalog facility defined in section [Catalog Facility](#).

8.6. javax.xml.ws.WebEndpoint

The `WebEndpoint` annotation is specified on the `get`PortName()`` methods of a generated service class (see [Service and Port](#)). It is used to associate a get method with a specific `wsdl:port`, identified by its local name (a `NCName`).

Table 7.6: `WebEndpoint` properties.

Property	Description	Default
<code>name</code>	The local name of the port	""

8.6.1. Example

The following shows a WSDL extract and the resulting generated service class.

```

<!-- WSDL extract -->
<wsdl:service name="StockQuoteService">
  <wsdl:port name="StockQuoteHTTPPort" binding="StockQuoteHTTPBinding"/>
  <wsdl:port name="StockQuoteSMTPPort" binding="StockQuoteSMTPBinding"/>
</wsdl:service>

// Generated Service Class
@WebServiceClient(name="StockQuoteService",
    targetNamespace="...",
    wsdlLocation="...")
public class StockQuoteService extends javax.xml.ws.Service {
    public StockQuoteService() {
        super(wsdlLocation_fromAnnotation, serviceName_fromAnnotation);
    }

    public StockQuoteService(String wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    // Other StockQuoteService constructors
    ...

    @WebEndpoint(name="StockQuoteHTTPPort")
    public StockQuoteProvider getStockQuoteHTTPPort() {
        return super.getPort(portName, StockQuoteProvider.class);
    }

    @WebEndpoint(name="StockQuoteHTTPPort")
    public StockQuoteProvider getStockQuoteHTTPPort(WebServiceFeature... f) {
        return super.getPort(portName, StockQuoteProvider.class, f);
    }

    @WebEndpoint(name="StockQuoteSMTPPort")
    public StockQuoteProvider getStockQuoteSMTPPort() {
        return super.getPort(portName, StockQuoteProvider.class);
    }

    @WebEndpoint(name="StockQuoteSMTPPort")
    public StockQuoteProvider getStockQuoteSMTPPort(WebServiceFeature... f) {
        return super.getPort(portName, StockQuoteProvider.class, f);
    }
}

```

8.7. javax.xml.ws.WebServiceProvider

The `WebServiceProvider` annotation is specified on classes that implement a strongly typed `javax.xml.ws.Provider`. It is used to declare that a class that satisfies the requirements for a provider (see `javax.xml.ws.Provider`) does indeed define a Web service endpoint, much like the `WebService` annotation does for SEI-based endpoints.

The `WebServiceProvider` and `WebService` annotations are mutually exclusive.

Conformance (WebServiceProvider and WebService): A class annotated with the `WebServiceProvider` annotation MUST NOT carry a `WebService` annotation.

Table 7.7: `WebServiceProvider` properties.

Property	Description	Default
<code>wsdlLocation</code>	The URL for the WSDL description	""
<code>serviceName</code>	The name of the service	""
<code>portName</code>	The name of the port	""
<code>targetNamespace</code>	The target namespace for the service	""

When resolving the URL specified as the `wsdlLocation` element or any document it may transitively reference, a JAX-WS implementation MUST use the catalog facility defined in section [Catalog Facility](#).

8.8. javax.xml.ws.BindingType

The `BindingType` annotation is applied to an endpoint implementation class. It specifies the binding to use when publishing an endpoint of this type.

Table 7.8: `BindingType` properties.

Property	Description	Default
<code>value</code>	The binding ID (a URI)	""

The default binding for an endpoint is the SOAP 1.1/HTTP one (see chapter [SOAP Binding](#)).

8.9. javax.xml.ws.WebServiceRef

The `WebServiceRef` annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the `javax.annotation.Resource` annotation in JSR-250 [\[35\]](#).

The injected references of `WebServiceRef` annotation are not guaranteed to be thread safe. If the references are accessed by multiple threads, usual synchronization techniques can be used to support

multiple threads.

The `WebServiceRef` annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification [36].

The injected references of `WebServiceRef` annotation can be configured with the corresponding features of annotations annotated with `WebServiceFeatureAnnotation` annotations. For example, a SEI reference can be configured with the MTOM web service feature using `@MTOM`. Similarly, a generated service reference can be configured with web service features using the corresponding `WebServiceFeatureAnnotation` annotations. JAX-WS specification defines three standard features `AddressingFeature`, `MTOMFeature` and `RespectBindingFeature` that can be used while creating proxy instances. However, there are no standard features that can be used while creating service instances in the current specification. A JAX-WS implementation may define its own features but they will be non-portable across all JAX-WS implementations.

If a JAX-WS implementation encounters an unsupported or unrecognized annotation annotated with the `WebServiceFeatureAnnotation` that is specified with `@WebServiceRef`, an error MUST be given.

Table 7.9: `WebServiceRef` properties.

Property	Description	Default
<code>name</code>	The name identifying the Web service reference.	""
<code>wsdlLocation</code>	A URL pointing to the location of the WSDL document for the service being referred to.	""
<code>type</code>	The resource type as a Java class object	<code>Object.class</code>
<code>value</code>	The service type as a Java class object	<code>Service.class</code>
<code>mappedName</code>	A product specific name that this resource should be mapped to.	""
<code>lookup</code>	A portable JNDI lookup name that resolves to the target web service reference.	""

The name of the resource, as defined by the `name` element (or defaulted) is a name that is local to the application component using the resource.

The name can be absolute JNDI name(with a logical namespace) or relative to the JNDI `java:comp/env` namespace.

Many application servers provide a way to map these local names to names of resources known to the application server. This `mappedName` is often a global JNDI name, but may be a name of any form. Application servers are not required to support any particular form or type of mapped name, nor the

ability to use mapped names. A mapped name is product-dependent and often installation-dependent. No use of a mapped name is portable.

A defined reference can be resolved using a portable JNDI name provided by `lookup` element. In this case, it is an error if there are any circular dependencies between entries of references. Similarly, it is an error if looking up the specified JNDI name results in a resource whose type is not compatible with the reference being created. Since this `'lookup'` functionality is just resolving to an already defined reference, only `'name'` can be specified with `lookup` (doesn't require any other metadata like `wsdlLocation` etc.).

There are two uses to the `WebServiceRef` annotation:

1. To define a reference whose type is a generated service class. In this case, the `type` and `value` element will both refer to the generated service class type. Moreover, if the reference type can be inferred by the field/method declaration the annotation is applied to, the `type` and `value` elements MAY have the default value (`Object.class`, that is). If the type cannot be inferred, then at least the `type` element MUST be present with a non-default value.
2. To define a reference whose type is a SEI. In this case, the `type` element MAY be present with its default value if the type of the reference can be inferred from the annotated field/method declaration, but the `value` element MUST always be present and refer to a generated service class type (a subtype of `javax.xml.ws.Service`).

The `wsdlLocation` element, if present, overrides the WSDL location information specified in the `WebService` annotation of the referenced generated service class.

When resolving the URI specified as the `wsdlLocation` element or any document it may transitively reference, a JAX-WS implementation MUST use the catalog facility defined in section [Catalog Facility](#).

8.9.1. Example

The following shows both uses of the `WebServiceRef` annotation.


```
// Generated Service Class

@WebServiceClient(name="StockQuoteService",
    targetNamespace="...",
    wsdlLocation="...")
public class StockQuoteService extends javax.xml.ws.Service {
    ...

    @WebEndpoint(name="StockQuoteHTTPPort")
    StockQuoteProvider getStockQuoteHTTPPort() { ... };

    @WebEndpoint(name="StockQuoteHTTPPort")
    StockQuoteProvider getStockQuoteHTTPPort(WebServiceFeature ... f) { ... };

    @WebEndpoint(name="StockQuoteSMTPPort")
    StockQuoteProvider getStockQuoteSMTPPort() { ... };

    @WebEndpoint(name="StockQuoteSMTPPort")
    StockQuoteProvider getStockQuoteSMTPPort(WebServiceFeature ... f) { ... };
}

// Generated SEI

@WebService(name="StockQuoteProvider",
    targetNamespace="...")
public interface StockQuoteProvider {
    Double getStockQuote(String ticker);
}

// Sample client code

@Stateless
public class ClientComponent {

    // WebServiceRef using the generated service interface type
    @WebServiceRef
    public StockQuoteService stockQuoteService;

    // WebServiceRef using the SEI type
    // stockQuoteProvider proxy is configured with MTOM feature
    @MTOM
    @WebServiceRef(StockQuoteService.class)
    private StockQuoteProvider stockQuoteProvider;

    // other methods go here...
}
```

8.10. javax.xml.ws.WebServiceRefs

The `WebServiceRefs` annotation is used to declare multiple references to Web services on a single class. It is necessary to work around the limitation against specifying repeated annotations of the same type on any given class, which prevents listing multiple `javax.ws.WebServiceRef` annotations one after the other. This annotation follows the resource pattern exemplified by the `javax.annotation.Resources` annotation in JSR-250 [35].

Since no name and type can be inferred in this case, each `WebServiceRef` annotation inside a `WebServiceRefs` MUST contain `name` and `type` elements with non-default values.

The `WebServiceRef` annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification [36].

Table 7.10: `WebServiceRefs` properties.

Property	Description	Default
<code>value</code>	An array of <code>WebServiceRef</code> annotations, each defining a web service reference.	<code>{}</code>

There is no way to associate web service features with the injected instances of this annotation. If an instance needs to be configured with the web service features, use `@WebServiceRef` to inject the resource along with its features.

8.10.1. Example

The following shows how to use the `WebServiceRefs` annotation to declare at the class level two web service references. The first one uses the SEI type, while the second one uses a generated service class type.

```
@WebServiceRefs({@WebServiceRef(name="accounting"
                                type=AccountingPortType.class,
                                value=AccountingService.class),
                 @WebServiceRef(name="payroll",
                                type=PayrollService.class)})

@Stateless
public MyComponent {

    // methods using the declared resources go here...
}
```

8.11. Annotations Defined by JSR-181

In addition to the annotations defined in the preceding sections, JAX-WS 2.0 uses several annotations defined by JSR-181.

Conformance (JSR-181 conformance): A JAX-WS 2.0 implementation **MUST** be conformant to the JAX-WS profile of JSR-181 1.1 [16].

As a convenience to the reader, the following sections reproduce the definition of the JSR-181 annotations applicable to JAX-WS.

8.11.1. javax.jws.WebService

```
@Target({TYPE})
public @interface WebService {
    String name() default "";
    String targetNamespace() default "";
    String serviceName() default "";
    String wsdlLocation() default "";
    String endpointInterface() default "";
    String portName() default "";
};
```

Consistently with the URI resolution process in JAX-WS, when resolving the URI specified as the `wsdlLocation` element or any document it may transitively reference, a JAX-WS implementation **MUST** use the catalog facility defined in section [Catalog Facility](#).

8.11.2. javax.jws.WebMethod

```
@Target({METHOD})
public @interface WebMethod {
    String operationName() default "";
    String action() default "";
    boolean exclude() default false;
};
```

8.11.3. javax.jws.Oneway

```
@Target({METHOD})
public @interface Oneway {
};
```

8.11.4. javax.jws.WebParam

```
@Target({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 "";
};
```

8.11.5. javax.jws.WebResult

```
@Target({METHOD})
public @interface WebResult {
    String name() default "return";
    String targetNamespace() default "";
    boolean header() default false;
    String partName() default "";
};
```

8.11.6. javax.jws.SOAPBinding

```
@Target({TYPE, 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;
}
```

8.11.7. javax.jws.HandlerChain

```

@Target({TYPE})
public @interface HandlerChain {
    String file();
    String name() default "";
}

```

8.12. javax.xml.ws.Action

The **Action** annotation is applied to the methods of a SEI. It is used to specify the **input**, **output**, **fault** WS-Addressing Action values associated with the annotated method.

For such a method, the mapped operation in the generated WSDL's **wsam:Action** attribute on the WSDL **input**, **output** and **fault** messages of the WSDL operation is based upon which attributes of the Action annotation have been specified. The **wsam:Action** computation algorithm is specified in section [wsam:Action Computation Algorithm](#)

Table 7.11: **FaultAction** properties.

Property	Description	Default
fault	Array of FaultAction for the wsdl:fault s of the operation	""
input	Action for the wsdl:input of the operation	""
output	Action for the wsdl:output of the operation	""

8.13. javax.xml.ws.FaultAction

The **FaultAction** annotation is used within the **Action** annotation to specify the WS-Addressing Action of a service specific exception as defined by section [Service Specific Exception](#).

The **wsam:Action** attribute value in the fault message in the generated WSDL operation mapped for an exception class is equal to the corresponding **value** in the FaultAction. The **wsam:Action** computation algorithm is specified in section [wsam:Action Computation Algorithm](#)

Table 7.12: **FaultAction** properties.

Property	Description	Default
value	Action for the wsdl:fault of the operation	""
output	Name of the exception class	no defaults required property

8.14. javax.xml.ws.spi.WebServiceFeatureAnnotation

The `WebServiceFeatureAnnotation` is a meta-annotation used by a JAX-WS implementation to identify other annotations as `WebServiceFeature`'s. JAX-WS provides the following annotations as `WebServiceFeatures`: `javax.xml.ws.soap.Addressing`, `javax.xml.ws.soap.MTOM`, and `javax.xml.ws.RespectBinding`. If a JAX-WS implementation encounters an annotation annotated with the `WebServiceFeatureAnnotation` that it does not support or recognize an ERROR MUST be given.

Table 7.13: `WebServiceFeatureAnnotation` properties.

Property	Description	Default
<code>id</code>	Unique identifier for the <code>WebServiceFeature</code> represented by the annotated annotation.	No defaults required property
<code>bean</code>	The class name of a derived <code>WebServiceFeature</code> class associated with the annotated annotation.	No defaults required property

The following shows how the `Addressing` annotation uses the `WebServiceFeatureAnnotation` meta-annotation.

```
@WebServiceFeatureAnnotation(id=AddressingFeature.ID,
                             bean=AddressingFeature.class)
public @interface Addressing {
    /**
     * Specifies if this feature is enabled or disabled.
     */
    boolean enabled() default true;

    /**
     * Property to determine whether WS-Addressing
     * headers MUST be present on incoming messages.
     */
    boolean required() default false;
}
```

8.14.1. javax.xml.ws.soap.Addressing

The `Addressing` annotation is applied to an endpoint implementation class and to an injected web service proxy reference. It is used to control the use of WS-Addressing[26][37][27]. It corresponds with the `AddressingFeature` described in section `javax.xml.ws.soap.AddressingFeature`.

Table 7.14: `Addressing` properties.

Property	Description	Default
<code>enabled</code>	Specifies if WS-Addressing is enabled or not.	true
<code>required</code>	Specifies Addressing headers MUST be present on incoming messages.	false
<code>responses</code>	If addressing is enabled, this property determines if endpoint requires the use of anonymous responses, or non-anonymous responses, or all.	<code>Responses.ALL</code>

The generated WSDL must indicate the use of addressing as specified in the section [Service and Ports](#). The runtime requirements of enabling addressing for SOAP binding are specified in section [Addressing](#).

8.14.2. javax.xml.ws.soap.MTOM

The `MTOM` annotation is applied to an endpoint implementation class and to an injected web service proxy reference. It is used to control the use of MTOM. It corresponds to the `MTOMFeature` described in section [javax.xml.ws.soap.MTOMFeature](#).

Table 7.15: `MTOM` properties.

Property	Description	Default
<code>enabled</code>	Specifies if MTOM is enabled or not.	true
<code>threshold</code>	Specifies the size in bytes that binary data SHOULD be before being sent as an attachment.	0

8.14.3. javax.xml.ws.RespectBinding

The `RespectBinding` annotation is applied to an endpoint implementation class and to an injected web service proxy reference

1. It is used to control whether a JAX-WS implementation MUST respect/honor the contents of the `wsdl:binding` associated with an endpoint. It has a corresponding `RespectBindingFeature` described in section [javax.xml.ws.RespectBindingFeature](#).

Table 7.16: `RespectBinding` properties.

Property	Description	Default
----------	-------------	---------

<code>enabled</code>	Specifies whether the wsdl:binding must be respected or not.	<code>true</code>
----------------------	--	-------------------

Chapter 9. Customizations

This chapter describes a standard customization facility that can be used to customize the WSDL 1.1 to Java binding defined in section [WSDL 1.1 to Java Mapping](#).

9.1. Binding Language

JAX-WS 2.0 defines an XML-based language that can be used to specify customizations to the WSDL 1.1 to Java binding. In order to maintain consistency with JAXB, we call it a *binding language*. Similarly, customizations will hereafter be referred to as *binding declarations*.

All XML elements defined in this section belong to the <http://java.sun.com/xml/ns/jaxws> namespace. For clarity, the rest of this section uses qualified element names exclusively. Wherever it appears, the `jaxws` prefix is assumed to be bound to the <http://java.sun.com/xml/ns/jaxws> namespace name.

The binding language is extensible. Extensions are expressed using elements and/or attributes whose namespace name is different from the one used by this specification.

[Standard binding declarations] The <http://java.sun.com/xml/ns/jaxws> namespace is reserved for standard JAX-WS binding declarations. Implementations **MUST** support all standard JAX-WS binding declarations. Implementation-specific binding declaration extensions **MUST NOT** use the <http://java.sun.com/xml/ns/jaxws> namespace.

[Binding language extensibility] Implementations **MUST** ignore unknown elements and attributes appearing inside a binding declaration whose namespace name is not the one specified in the standard, i.e. <http://java.sun.com/xml/ns/jaxws>.

9.2. Binding Declaration Container

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* (see [External Binding File](#)). The second approach consists in embedding binding declarations directly inside a WSDL document (see [Embedded Binding Declarations](#)).

In either case, the `jaxws:bindings` element is used as a container for JAX-WS binding declarations. It contains a (possibly empty) list of binding declarations, in any order.

```
<jaxws:bindings wsdlLocation="xs:anyURI"?  
                node="xs:string"?  
                version="string"?>  
    ...binding declarations...  
</jaxws:bindings>
```

Figure 8.1: Syntax of the binding declaration container

Semantics

@wsdlLocation

A URI pointing to a WSDL file establishing the scope of the contents of this binding declaration. It MUST NOT be present if the `jaxws:bindings` element is used as an extension inside a WSDL document or one of its ancestor `jaxws:bindings` elements already contains this attribute.

@node

An XPath expression pointing to the element in the WSDL file in scope that this binding declaration is attached to. It MUST NOT be present if the `jaxws:bindings` appears inside a WSDL document.

@version

A version identifier. It MUST NOT appear on `jaxws:bindings` elements which have any `jaxws:bindings` ancestors (i.e. on non top-level binding declarations).

For the JAX-WS 2.0 specification, the version identifier, if present, MUST be `2.0`. If the `@version` attribute is absent, it will implicitly be assumed to be `2.0`.

9.3. Embedded Binding Declarations

An embedded binding declaration is specified by using the `jaxws:bindings` element as a WSDL extension. Embedded binding declarations MAY appear on any of the elements in the WSDL 1.1 namespace that accept extension elements, per the schema for the WSDL 1.1 namespace as amended by the WS-I Basic Profile 1.1[20].

A binding declaration embedded in a WSDL document can only affect the WSDL element it extends. When a `jaxws:bindings` element is used as a WSDL extension, it MUST NOT have a `node` attribute. Moreover, it MUST NOT have an element whose qualified name is `jaxws:bindings` among its children.

9.3.1. Example

Figure [8.2] shows a WSDL document containing binding declaration extensions. For JAXB annotations, it assumes that the prefix `jaxb` is bound to the namespace name `http://java.sun.com/xml/ns/jaxb`.

9.4. External Binding File

The `jaxws:bindings` element MAY appear as the root element of a XML document. Such a document is called an *external binding file*.

An external binding file specifies bindings for a given WSDL document. The WSDL document in question is identified via the mandatory `wsdlLocation` attribute on the root `jaxws:bindings` element in the document.

```
<wsdl:definitions targetNamespace="..." xmlns:tns="..." xmlns:stns="...">
  <wsdl:types>
```

```

<xs:schema targetNamespace="http://example.org/bar">
  <xs:annotation>
    <xs:appinfo>
      <jaxb:bindings>
        ...some JAXB binding declarations...
      </jaxb:bindings>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="setLastTradePrice">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="tickerSymbol" type="xs:string"/>
        <xs:element name="lastTradePrice" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="setLastTradePriceResponse">
    <xs:complexType>
      <xs:sequence/>
    </xs:complexType>
  </xs:element>
</xs:schema>
</wsdl:types>

<wsdl:message name="setLastTradePrice">
  <wsdl:part name="setPrice" element="tns:setLastTradePrice"/>
</wsdl:message>

<wsdl:message name="setLastTradePriceResponse">
  <wsdl:part name="setPriceResponse" type="tns:setLastTradePriceResponse"/>
</wsdl:message>

<wsdl:portType name="StockQuoteUpdater">
  <wsdl:operation name="setLastTradePrice">
    <wsdl:input message="tns:setLastTradePrice"/>
    <wsdl:output message="tns:setLastTradePriceResponse"/>
    <jaxws:bindings>
      <jaxws:method name="updatePrice"/>
    </jaxws:bindings>
  </wsdl:operation>
  <jaxws:bindings>
    <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
  </jaxws:bindings>
</wsdl:portType>

<jaxws:bindings>
  <jaxws:package name="com.acme.foo"/>
  ...additional binding declarations...

```

```
</jaxws:bindings>
</wsdl:definitions>
```

In an external binding file, `jaxws:bindings` elements MAY appear as non-root elements, e.g. as a child or descendant of the root `jaxws:bindings` element. In this case, they MUST carry a `node` attribute identifying the element in the WSDL document they annotate. The root `jaxws:bindings` element implicitly contains a `node` attribute whose value is `//`, i.e. selecting the root element in the document. An XPath expression on a non-root `jaxws:bindings` element selects zero or more nodes from the set of nodes selected by its parent `jaxws:bindings` element.

External binding files are semantically equivalent to embedded binding declarations (see [Embedded Binding Declarations](#)). When a JAX-WS implementation processes a WSDL document for which there is an external binding file, it MUST operate as if all binding declarations specified in the external binding file were instead specified as embedded declarations on the nodes in the in the WSDL document they target. It is an error if, upon embedding the binding declarations defined in one or more external binding files, the resulting WSDL document contains conflicting binding declarations.

Conformance (Multiple binding files): Implementations MUST support specifying any number of external JAX-WS and JAXB binding files for processing in conjunction with at least one WSDL document.

Please refer to section [Using JAXB Binding Declarations](#) for more information on processing JAXB binding declarations.

9.4.1. Example

Figures [\[8.3\]](#) and [\[8.4\]](#) show an example external binding file and WSDL document respectively that express the same set of binding declarations as the WSDL document in [Example](#).

```
<jaxws:bindings wsdlLocation="http://example.org/foo.wsdl">
  <jaxws:package name="com.acme.foo"/>
  <jaxws:bindings
    node="wsdl:types/xs:schema[targetNamespace='http://example.org/bar']">
    <jaxb:bindings>
      ...some JAXB binding declarations...
    </jaxb:bindings>
  </jaxws:bindings>
  <jaxws:bindings node="wsdl:portType[@name='StockQuoteUpdater']">
    <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
    <jaxws:bindings node="wsdl:operation[@name='setLastTradePrice']">
      <jaxws:method name="updatePrice"/>
    </jaxws:bindings>
  </jaxws:bindings>
  ...additional binding declarations....
</jaxws:bindings>
```

Figure 8.3: Sample external binding file for WSDL in figure [8.4]

9.5. Using JAXB Binding Declarations

It is possible to use JAXB binding declarations in conjunction with JAX-WS.

The JAXB 2.0 bindings element, henceforth referred to as `jaxb:bindings`, MAY appear as an annotation inside a schema document embedded in a WSDL document, i.e. as a descendant of a `xs:schema` element whose parent is the `wSDL:types` element. It affects the data binding as specified by JAXB 2.0.

```
<wSDL:definitions targetNamespace="..." xmlns:tns="..." xmlns:stns="...">
  <wSDL:types>
    <xs:schema targetNamespace="http://example.org/bar">
      <xs:element name="setLastTradePrice">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="tickerSymbol" type="xs:string"/>
            <xs:element name="lastTradePrice" type="xs:float"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="setLastTradePriceResponse">
        <xs:complexType>
          <xs:sequence/>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wSDL:types>

  <wSDL:message name="setLastTradePrice">
    <wSDL:part name="setPrice" element="stns:setLastTradePrice"/>
  </wSDL:message>

  <wSDL:message name="setLastTradePriceResponse">
    <wSDL:part name="setPriceResponse"
      type="stns:setLastTradePriceResponse"/>
  </wSDL:message>

  <wSDL:portType name="StockQuoteUpdater">
    <wSDL:operation name="setLastTradePrice">
      <wSDL:input message="tns:setLastTradePrice"/>
      <wSDL:output message="tns:setLastTradePriceResponse"/>
    </wSDL:operation>
  </wSDL:portType>
</wSDL:definitions>
```

Figure 8.4: WSDL document referred to by external binding file in figure [8.3]

Additionally, `jaxb:bindings` MAY appear inside a JAX-WS external binding file as a child of a `jaxws:bindings` element whose `node` attribute points to a `xs:schema` element inside a WSDL document. When the schema is processed, the outcome MUST be as if the `jaxb:bindings` element was inlined inside the schema document as an annotation on the schema component.

While processing a JAXB binding declaration (i.e. a `jaxb:bindings` element) for a schema document embedded inside a WSDL document, all XPath expressions that appear inside it MUST be interpreted as if the containing `xs:schema` element was the root of a standalone schema document.

Editors Note 8.1 *This last requirement ensures that JAXB processors don't have to be extended to incorporate knowledge of WSDL. In particular, it becomes possible to take a JAXB binding file and embed it in a JAX-WS binding file as-is, without fixing up all its XPath expressions, even in the case that the XML Schema the JAXB binding file refers to was embedded in a WSDL.*

9.6. Scoping of Bindings

Binding declarations are scoped according to the parent-child hierarchy in the WSDL document. For instance, when determining the value of the `jaxws:enableWrapperStyle` customization parameter for a portType operation, binding declarations MUST be processed in the following order, according to the element they pertain to: (1) the portType operation in question, (2) its parent portType, (3) the definitions element.

Tools MUST NOT ignore binding declarations. It is an error if upon applying all the customizations in effect for a given WSDL document, any of the generated Java source code artifacts does not contain legal Java syntax. In particular, it is an error to use any reserved keywords as the name of a Java field, method, type or package.

9.7. Standard Binding Declarations

The following sections detail the predefined binding declarations, classified according to the WSDL element they're allowed on. All these declarations reside in the <http://java.sun.com/xml/ns/jaxws> namespace.

9.7.1. Definitions

The following binding declarations MAY appear in the context of a WSDL document, either as an extension to the `wSDL:definitions` element or in an external binding file at a place where there is a WSDL document in scope.

```

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

<jaxws:enableWrapperStyle>?
  xs:boolean
</jaxws:enableWrapperStyle>

<jaxws:enableAsyncMapping>?
  xs:boolean
</jaxws:enableAsyncMapping>

<jaxws:enableMIMEContent>?
  xs:boolean
</jaxws:enableMIMEContent>

```

9.7.1.1. Semantics

package/@name

Name of the Java package for the targetNamespace of the parent `wsdl:definitions` element.

package/javadoc/text()

Package-level javadoc string.

enableWrapperStyle

If present with a boolean value of `true` (resp. `false`), wrapper style is enabled (resp. disabled) by default for all operations.

enableAsyncMapping

If present with a boolean value of `true` (resp. `false`), asynchronous mappings are enabled (resp. disabled) by default for all operations.

enableMIMEContent

If present with a boolean value of `true` (resp. `false`), use of the `mime:content` information is enabled (resp. disabled) by default for all operations.

The `enableWrapperStyle` declaration only affects operations that qualify for the wrapper style per the JAX-WS specification. By default, this declaration is `true`, i.e. wrapper style processing is turned on by default for all qualified operations, and must be disabled by using a `jaxws:enableWrapperStyle` declaration with a value of `false` in the appropriate scope.

9.7.2. PortType

The following binding declarations MAY appear in the context of a WSDL portType, either as an extension to the `wsdl:portType` element or with a `node` attribute pointing at one.

```

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

<jaxws:enableWrapperStyle>?
  xs:boolean
</jaxws:enableWrapperStyle>

<jaxws:enableAsyncMapping>xs:boolean</jaxws:enableAsyncMapping>?

```

9.7.2.1. Semantics

class/@name

Fully qualified name of the generated service endpoint interface corresponding to the parent `wsdl:portType`.

class/javadoc/text()

Class-level javadoc string.

enableWrapperStyle

If present with a boolean value of `true` (resp. `false`), wrapper style is enabled (resp. disabled) by default for all operations in this `wsdl:portType`.

enableAsyncMapping

If present with a boolean value of `true` (resp. `false`), asynchronous mappings are enabled (resp. disabled) by default for all operations in this `wsdl:portType`.

9.7.3. PortType Operation

The following binding declarations MAY appear in the context of a WSDL portType operation, either as an extension to the `wsdl:portType/wsdl:operation` element or with a `node` attribute pointing at one.


```

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

<jaxws:enableWrapperStyle>?
  xs:boolean
</jaxws:enableWrapperStyle>

<jaxws:enableAsyncMapping>?
  xs:boolean
</jaxws:enableAsyncMapping>

<jaxws:parameter part="xs:string"
  childElementName="xs:QName"?
  name="xs:string"/>*

```

9.7.3.1. Semantics

method/@name

Name of the Java method corresponding to this `wsdl:operation`.

method/javadoc/text()

Method-level javadoc string.

enableWrapperStyle

If present with a boolean value of `true` (resp. `false`), wrapper style is enabled (resp. disabled) by default for this `wsdl:operation`.

enableAsyncMapping

If present with a boolean value of `true`, asynchronous mappings are enabled by default for this `wsdl:operation`.

parameter/@part

A XPath expression identifying a `wsdl:part` child of a `wsdl:message`.

parameter/@childElementName

The qualified name of a child element information item of the global type definition or global element declaration referred to by the `wsdl:part` identified by the previous attribute.

parameter/@name

The name of the Java formal parameter corresponding to the parameter identified by the previous two attributes.

It is an error if two parameters that do not correspond to the same Java formal parameter are assigned the same name, or if a part/element that corresponds to the Java method return value is assigned a

name.

9.7.4. PortType Fault Message

The following binding declarations MAY appear in the context of a WSDL portType operation's fault message, either as an extension to the `wSDL:portType/wSDL:operation/wSDL:fault` element or with a `node` attribute pointing at one.

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

9.7.4.1. Semantics

Fully qualified name of the generated exception class for this fault.

Class-level javadoc string.

It is an error if faults that refer to the same `wSDL:message` element are mapped to exception classes with different names.

9.7.5. Binding

The following binding declarations MAY appear in the context of a WSDL binding, either as an extension to the `wSDL:binding` element or with a `node` attribute pointing at one.

```
<jaxws:enableMIMEContent>?
  xs:boolean
</jaxws:enableMIMEContent>
```

9.7.5.1. Semantics

`enableMIMEContent`

If present with a boolean value of `true` (resp. `false`), use of the `mime:content` information is enabled (resp. disabled) for all operations in this binding.

9.7.6. Binding Operation

The following binding declarations MAY appear in the context of a WSDL binding operation, either as an extension to the `wSDL:binding/wSDL:operation` element or with a `node` attribute pointing at one.

```

<jaxws:enableMIMEContent>?
  xs:boolean
</jaxws:enableMIMEContent>

<jaxws:parameter part="xs:string"
  childElementName="xs:QName"?
  name="xs:string"/>*

<jaxws:exception part="xs:string">*
  <jaxws:class name="xs:string">?
    <jaxws:javadoc>xs:string</jaxws:javadoc>?
  </jaxws:class>
</jaxws:exception>

```

9.7.6.1. Semantics

enableMIMEContent

If present with a boolean value of **true** (resp. **false**), use of the **mime:content** information is enabled (resp. disabled) for this operation.

parameter/@part

A XPath expression identifying a **wSDL:part** child of a **wSDL:message**.

parameter/@childElementName

The qualified name of a child element information item of the global type definition or global element declaration referred to by the **wSDL:part** identified by the previous attribute.

parameter/@name

The name of the Java formal parameter corresponding to the parameter identified by the previous two attributes. The parameter in question MUST correspond to a **soap:header** extension.

9.7.7. Service

The following binding declarations MAY appear in the context of a WSDL service, either as an extension to the **wSDL:service** element or with a **node** attribute pointing at one.

```

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

```

9.7.7.1. Semantics

class/@name

Fully qualified name of the generated service class.

class/javadoc/text()

Class-level javadoc string.

9.7.8. Port

The following binding declarations MAY appear in the context of a WSDL service, either as an extension to the `wsdl:port` element or with a `node` attribute pointing at one.

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

<jaxws:provider/>?
```

9.7.8.1. Semantics

method/@name

The name of the generated port getter method.

method/javadoc/text()

Method-level javadoc string.

provider

This binding declaration specifies that the annotated port will be used with the `javax.xml.ws.Provider` interface.

A port annotated with a `jaxws:provider` binding declaration is treated specially. No service endpoint interface will be generated for it, since the application code will use in its lieu the `javax.xml.ws.Provider` interface. Additionally, the port getter method on the generated service interface will be omitted.

Editors Note 8.2 *Omitting a `getXYZPort()` method is necessary for consistency, because if it existed it would specify the non-existing SEI type as its return type.*

Chapter 10. Handler Framework

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-WS runtime system. This chapter describes the handler framework in detail.

Conformance (Handler framework support): An implementation **MUST** support the handler framework.

10.1. Architecture

The handler framework is implemented by a JAX-WS protocol binding in both client and server side runtimes. Proxies, and **Dispatch** instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols (see figure [\[handarchfig\]](#)). Protocol bindings can extend the handler framework to provide protocol specific functionality; chapter [SOAP Binding](#) describes the JAX-WS SOAP binding that extends the handler framework with SOAP specific functionality.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

10.1.1. Types of Handler

JAX-WS 2.0 defines two types of handler:

Logical

Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler`.

Protocol

Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message. Protocol handlers are handlers that implement any interface derived from `javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

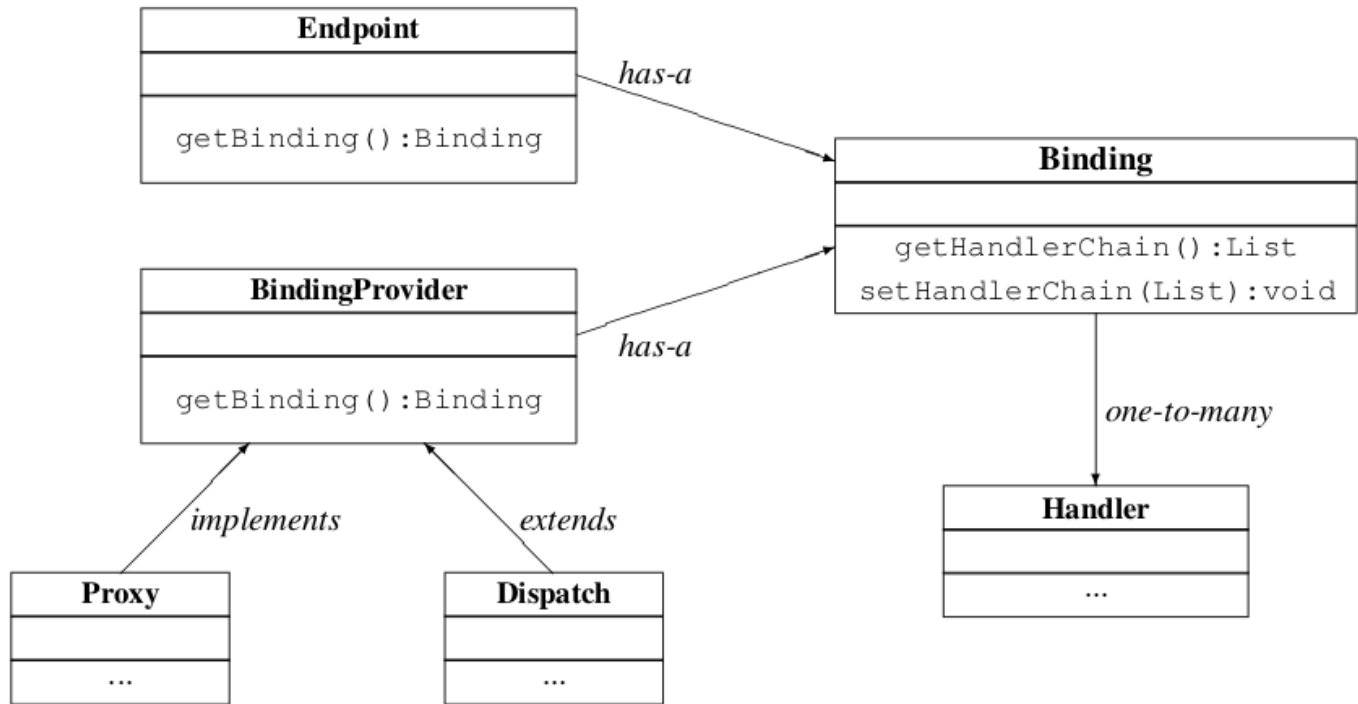


Figure 9.1: Handler architecture

Figure [9.2] shows the class hierarchy for handlers.

Handlers for protocols other than SOAP are expected to implement a protocol-specific interface that extends `javax.xml.ws.handler.Handler`.

10.1.2. Binding Responsibilities

The following subsections describe the responsibilities of the protocol binding when hosting a handler chain.

10.1.2.1. Handler and Message Context Management

The binding is responsible for instantiation, invocation, and destruction of handlers according to the rules specified in section [Processing Model](#). The binding is responsible for instantiation and management of message contexts according to the rules specified in section [Message Context](#)

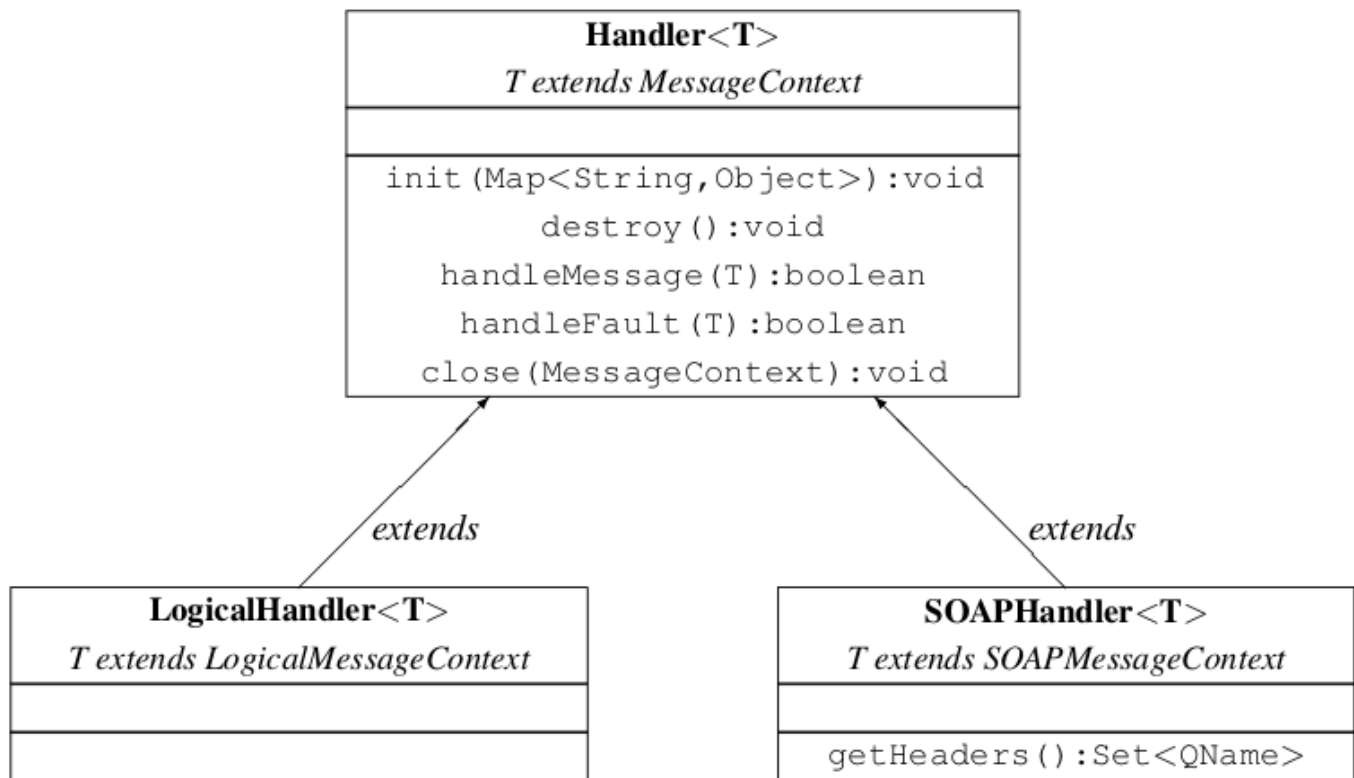


Figure 9.2: Handler class hierarchy

Conformance (Logical handler support): All binding implementations MUST support logical handlers (see section [Types of Handler](#)) being deployed in their handler chains.

Conformance (Other handler support): Binding implementations MAY support other handler types (see section [Types of Handler](#)) being deployed in their handler chains.

Conformance (Incompatible handlers): An implementation MUST throw `WebServiceException` when, at the time a binding provider is created, the handler chain returned by the configured `HandlerResolver` contains an incompatible handler.

Conformance (Incompatible handlers): Implementations MUST throw a `WebServiceException` when attempting to configure an incompatible handler using the `Binding.setHandlerChain` method.

10.1.2.2. Message Dispatch

The binding is responsible for dispatch of both outbound and inbound messages after handler processing. Outbound messages are dispatched using whatever means the protocol binding uses for communication. Inbound messages are dispatched to the binding provider. JAX-WS defines no standard interface between binding providers and their binding.

10.1.2.3. Exception Handling

The binding is responsible for catching runtime exceptions thrown by handlers and respecting any resulting message direction and message type change as described in section [Handler Execution](#).

Outbound exceptions [8: Outbound exceptions are exceptions thrown by a handler that result in the message direction being set to outbound according to the rules in section [Handler Execution](#).] are converted to protocol fault messages and dispatched using whatever means the protocol binding uses for communication. Specific protocol bindings describe the mechanism for their particular protocol, section [Exception Handling](#) describes the mechanism for the SOAP 1.1 binding. Inbound exceptions are passed to the binding provider.

10.2. Configuration

Handler chains may be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

10.2.1. Programmatic Configuration

JAX-WS only defines APIs for programmatic configuration of client side handler chains – server side handler chains are expected to be configured using deployment metadata.

10.2.1.1. `javax.xml.ws.handler.HandlerResolver`

A `Service` instance maintains a handler resolver that is used when creating proxies or `Dispatch` instances, known collectively as binding providers. During the creation of a binding provider, the handler resolver currently registered with a service is used to create a handler chain, which in turn is then used to configure the binding provider. A `Service` instance provides access to a `handlerResolver` property, via the `Service.getHandlerResolver` and `Service.setHandlerResolver` methods. A `HandlerResolver` implements a single method, `getHandlerChain`, which has one argument, a `PortInfo` object. The JAX-WS runtime uses the `PortInfo` argument to pass the `HandlerResolver` of the service, port and binding in use. The `HandlerResolver` may use any of this information to decide which handlers to use in constructing the requested handler chain.

When a `Service` instance is used to create an instance of a binding provider then the created instance is configured with the handler chain created by the `HandlerResolver` instance registered on the `Service` instance at that point in time.

Conformance (Handler chain snapshot): Changing the handler resolver configured for a `Service` instance MUST NOT affect the handlers on previously created proxies, or `Dispatch` instances.

10.2.1.2. Handler Ordering

The handler chain for a binding is constructed by starting with the handler chain as returned by the `HandlerResolver` for the service in use and sorting its elements so that all logical handlers precede all protocol handlers. In performing this operation, the order of handlers of any given type (logical or protocol) in the original chain is maintained. Figure [\[9.3\]](#) illustrates this.

Section [Handler Execution](#) describes how the handler order relates to the order of handler execution for inbound and outbound messages.

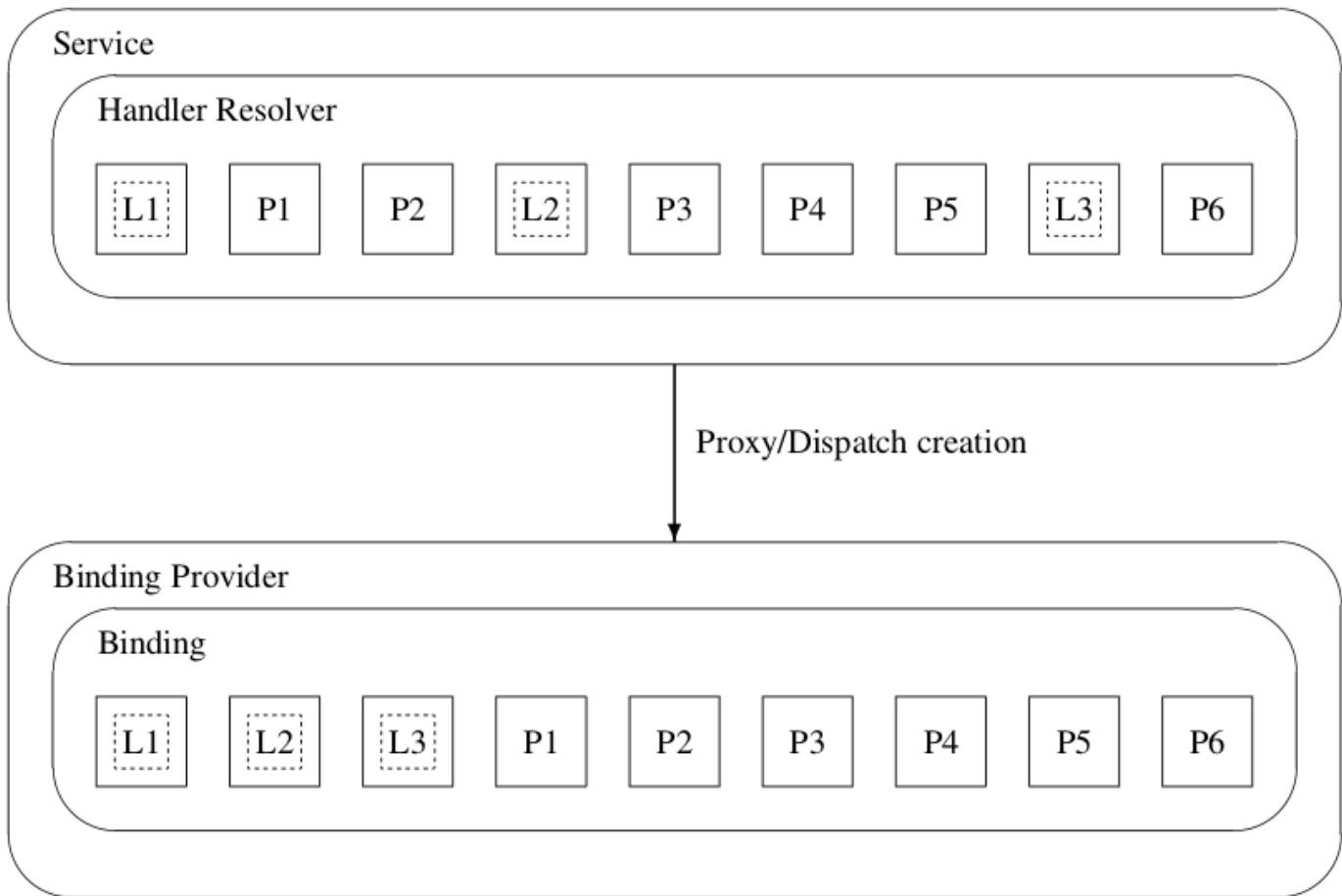


Figure 9.3: Handler ordering, Ln and Pn represent logical and protocol handlers respectively.

10.2.1.3. `javax.jws.HandlerChain` annotation

The `javax.jws.HandlerChain` annotation defined by JSR-181 [16] may be used to specify in a declarative way the handler chain to use for a service.

When used in conjunction with JAX-WS, the `name` element of the `HandlerChain` annotation, if present, MUST have the default value (the empty string).

In addition to appearing on a endpoint implementation class or a SEI, as specified by JSR-181, the `handlerChain` annotation MAY appear on a generated service class. In this case, it affects all the proxies and `Dispatch` instances created using any of the ports on the service.

Conformance (HandlerChain annotation): An implementation MUST support using the `HandlerChain` annotation on an endpoint implementation class, including a provider, on an endpoint interface and on a generated service class.

On the client, the `HandlerChain` annotation can be seen as a shorthand way of defining and installing a handler resolver (see [Handler Resolver](#)).

Conformance (Handler resolver for a HandlerChain annotation): For a generated service class (see [Service and Port](#)) which is annotated with a `HandlerChain` annotation, the default handler resolver

MUST return handler chains consistent with the contents of the handler chain descriptor referenced by the `HandlerChain` annotation.

Figure [9.4] shows an endpoint implementation class annotated with a `HandlerChain` annotation.

```
@WebService
@HandlerChain(file="sample_chain.xml")
public class MyService {
    ...
}
```

Figure 9.4: Use of the `HandlerChain` annotation

10.2.1.4. `javax.xml.ws.Binding`

The `Binding` interface is an abstraction of a JAX-WS protocol binding (see section [protocolbinding] for more details). As described above, the handler chain initially configured on an instance is a snapshot of the applicable handlers configured on the `Service` instance at the time of creation. `Binding` provides methods to manipulate the initially configured handler chain for a specific instance.

Conformance (Binding handler manipulation): Changing the handler chain on a `Binding` instance MUST NOT cause any change to the handler chains configured on the `Service` instance used to create the `Binding` instance.

10.2.2. Deployment Model

JAX-WS defines no standard deployment model for handlers. Such a model is provided by JSR 109[17] Implementing Enterprise Web Services.

10.3. Processing Model

This section describes the processing model for handlers within the handler framework.

10.3.1. Handler Lifecycle

In some cases, a JAX-WS implementation must instantiate handler classes directly, e.g. in a container environment or when using the `HandlerChain` annotation. When doing so, an implementation must invoke the handler lifecycle methods as prescribed in this section.

If an application does its own instantiation of handlers, e.g. using a handler resolver, then the burden of calling any handler lifecycle methods falls on the application itself. This should not be seen as inconsistent, because handlers are logically part of the application, so their contract will be known to the application developer.

The JAX-WS runtime system manages the lifecycle of handlers by invoking any methods of the handler

class annotated as lifecycle methods before and after dispatching requests to the handler itself.

The JAX-WS runtime system is responsible for loading the handler class and instantiating the corresponding handler object according to the instruction contained in the applicable handler configuration file or deployment descriptor.

The lifecycle of a handler instance begins when the JAX-WS runtime system creates a new instance of the handler class.

The runtime MUST then carry out any injections requested by the handler, typically via the `javax.annotation.Resource` annotation. After all the injections have been carried out, including in the case where no injections were requested, the runtime MUST invoke the method carrying a `javax.annotation.PostConstruct` annotation, if present. Such a method MUST satisfy the requirements in JSR-250 [35] for lifecycle methods (i.e. it has a void return type and takes zero arguments). The handler instance is then ready for use.

Conformance (Handler initialization): After injection has been completed, an implementation MUST call the lifecycle method annotated with `PostConstruct`, if present, prior to invoking any other method on a handler instance.

Once the handler instance is created and initialized it is placed into the `Ready` state. While in the `Ready` state the JAX-WS runtime system may invoke other handler methods as required.

The lifecycle of a handler instance ends when the JAX-WS runtime system stops using the handler for processing inbound or outbound messages. After taking the handler offline, a JAX-WS implementation SHOULD invoke the lifecycle method which carries a `javax.annotation.PreDestroy` annotation, if present, so as to permit the handler to clean up its resources. Such a method MUST satisfy the requirements in JSR-250 [35] for lifecycle methods

An implementation can only release handlers after the instance they are attached to, be it a proxy, a `Dispatch` object, an endpoint or some other component, e.g. a EJB object, is released. Consequently, in non-container environments, it is impossible to call the `PreDestroy` method in a reliable way, and handler instance cleanup must be left to finalizer methods and regular garbage collection.

Conformance (Handler destruction): In a managed environment, prior to releasing a handler instance, an implementation MUST call the lifecycle method annotated with `PreDestroy` method, if present, on any `Handler` instances which it instantiated.

The handler instance must release its resources and perform cleanup in the implementation of the `PreDestroy` lifecycle method. After invocation of the `PreDestroy` method(s), the handler instance will be made available for garbage collection.

10.3.2. Handler Execution

As described in section [Handler Ordering](#), a set of handlers is managed by a binding as an ordered list called a handler chain. Unless modified by the actions of a handler (see below) normal processing involves each handler in the chain being invoked in turn. Each handler is passed a message context

(see section [Message Context](#)) whose contents may be manipulated by the handler.

For outbound messages handler processing starts with the first handler in the chain and proceeds in the same order as the handler chain. For inbound messages the order of processing is reversed: processing starts with the last handler in the chain and proceeds in the reverse order of the handler chain. E.g., consider a handler chain that consists of six handlers $H_1 \dots H_6$ in that order: for outbound messages handler H_1 would be invoked first followed by H_2 , H_3 , ..., and finally handler H_6 ; for inbound messages H_6 would be invoked first followed by H_5 , H_4 , ..., and finally H_1 .

In the following discussion the terms next handler and previous handler are used. These terms are relative to the direction of the message, [Table 9.1](#) summarizes their meaning.

Handlers may change the direction of messages and the order of handler processing by throwing an exception or by returning `false` from `handleMessage` or `handleFault`. The following subsections describe each handler method and the changes to handler chain processing they may cause.

Message Direction	Term	Handler
Inbound	Next	H_{i-1}
	Previous	H_{i+1}
Outbound	Next	H_{i+1}
	Previous	H_{i-1}

Table 9.1: Next and previous handlers for handler H_i .

10.3.2.1. `handleMessage`

This method is called for normal message processing. Following completion of its work the `handleMessage` implementation can do one of the following:

Return `true`

This indicates that normal message processing should continue. The runtime invokes `handleMessage` on the next handler or dispatches the message (see section [Message Dispatch](#)) if there are no further handlers.

Return `false`

This indicates that normal message processing should cease. Subsequent actions depend on whether the message exchange pattern (MEP) in use requires a response to the `_message` currently being processed_ [9: For a request-response MEP, if the message direction is reversed during processing of a request message then the message becomes a response message. Subsequent handler processing takes this change into account.] or not: +

Response

The message direction is reversed, the runtime invokes `handleMessage` on the next [10: Next in this context means the next handler taking into account the message direction reversal] handler

or dispatches the message (see section [Message Dispatch](#)) if there are no further handlers.

No response

Normal message processing stops, `close` is called on each previously invoked handler in the chain, the message is dispatched (see section [Message Dispatch](#)).

Throw `ProtocolException` or a subclass

This indicates that normal message processing should cease. Subsequent actions depend on whether the MEP in use requires a response to the message currently being processed or not: +

Response

Normal message processing stops, fault message processing starts. The message direction is reversed, if the message is not already a fault message then it is replaced with a fault message [11: The handler may have already converted the message to a fault message, in which case no change is made.], and the runtime invokes `handleFault` on the nexthandler or dispatches the message (see section [Message Dispatch](#)) if there are no further handlers.

No response

Normal message processing stops, `close` is called on each previously invoked handler in the chain, the exception is dispatched (see section [Exception Handling](#)).

Throw any other runtime exception

This indicates that normal message processing should cease. Subsequent actions depend on whether the MEP in use includes a response to the message currently being processed or not: +

Response

Normal message processing stops, `close` is called on each previously invoked handler in the chain, the message direction is reversed, and the exception is dispatched (see section [Exception Handling](#)).

No response

Normal message processing stops, `close` is called on each previously invoked handler in the chain, the exception is dispatched (see section [Exception Handling](#)).

10.3.2.2. `handleFault`

Called for fault message processing, following completion of its work the `handleFault` implementation can do one of the following:

Return `true`

This indicates that fault message processing should continue. The runtime invokes `handleFault` on the next handler or dispatches the fault message (see section [Message Dispatch](#)) if there are no further handlers.

Return `false`

This indicates that fault message processing should cease. Fault message processing stops, `close` is called on each previously invoked handler in the chain, the fault message is dispatched (see section [Message Dispatch](#)).

Throw `ProtocolException` or a subclass

This indicates that fault message processing should cease. Fault message processing stops, `close` is called on each previously invoked handler in the chain, the exception is dispatched (see section [Exception Handling](#)).

Throw any other runtime exception

This indicates that fault message processing should cease. Fault message processing stops, `close` is called on each previously invoked handler in the chain, the exception is dispatched (see section [Exception Handling](#)).

10.3.2.3. `close`

A handler's `close` method is called at the conclusion of a message exchange pattern (MEP). It is called just prior to the binding dispatching the final message, fault or exception of the MEP and may be used to clean up per-MEP resources allocated by a handler. The `close` method is only called on handlers that were previously invoked via either `handleMessage` or `handleFault`.

Conformance (Invoking `close`): At the conclusion of an MEP, an implementation MUST call the `close` method of each handler that was previously invoked during that MEP via either `handleMessage` or `handleFault`.

Conformance (Order of `close` invocations): Handlers are invoked in the reverse order in which they were first invoked to handle a message according to the rules for normal message processing (see [Handler Execution](#)).

10.3.3. Handler Implementation Considerations

Handler instances may be pooled by a JAX-WS runtime system. All instances of a specific handler are considered equivalent by a JAX-WS runtime system and any instance may be chosen to handle a particular message. Different handler instances may be used to handle each message of an MEP. Different threads may be used for each handler in a handler chain, for each message in an MEP or any combination of the two. Handlers should not rely on thread local state to share information. Handlers should instead use the message context, see section [Message Context](#).

10.4. Message Context

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties.

Different types of handler are invoked with different types of message context. Sections [javax.xml.ws.handler.MessageContext](#) and [javax.xml.ws.handler.LogicalMessageContext](#) describe

`MessageContext` and `LogicalMessageContext` respectively. In addition, JAX-WS bindings may define a message context subtype for their particular protocol binding that provides access to protocol specific features. Section [SOAP Message Context](#) describes the message context subtype for the JAX-WS SOAP binding.

10.4.1. `javax.xml.ws.handler.MessageContext`

`MessageContext` is the super interface for all JAX-WS message contexts. It extends `Map<String, Object>` with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the `put` method to insert a property in the message context that one or more other handlers in the handler chain may subsequently obtain via the `get` method.

Properties are scoped as either `APPLICATION` or `HANDLER`. All properties are available to all handlers for an instance of an MEP on a particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during the execution of an MEP instance. `APPLICATION` scoped properties are also made available to client applications (see section [Configuration](#)) and service endpoint implementations. The default scope for a property is `HANDLER`.

Conformance (Message context property scope): Properties in a message context MUST be shared across all handler invocations for a particular instance of an MEP on any particular endpoint.

10.4.1.1. Standard Message Context Properties

[\[Table 9.2\]](#) lists the set of standard `MessageContext` properties.

The standard properties form a set of metadata that describes the context of a particular message. The property values may be manipulated by client applications, service endpoint implementations, the JAX-WS runtime or handlers deployed in a protocol binding. A JAX-WS runtime is expected to implement support for those properties shown as mandatory and may implement support for those properties shown as optional.

[\[Table 9.3\]](#) lists the standard `MessageContext` properties specific to the HTTP protocol. These properties are only required to be present when using an HTTP-based binding.

[\[Table 9.4\]](#) lists those properties that are specific to endpoints running inside a servlet container. These properties are only required to be present in the message context of an endpoint that is deployed inside a servlet container and uses an HTTP-based binding.

Table 9.2: Standard `MessageContext` properties.

Name	Type	Mandatory	Description
<code>javax.xml.ws.handler.message</code>			

<code>.outbound</code>	Boolean	Y	Specifies the message direction: <code>true</code> for outbound messages, <code>false</code> for inbound messages.
javax.xml.ws.binding.attachments			
<code>.inbound</code>	Map<String,DataHandler>	Y	A map of attachments to an inbound message. The key is a unique identifier for the attachment. The value is a <code>DataHandler</code> for the attachment data. Bindings describe how to carry attachments with messages.
<code>.outbound</code>	Map<String,DataHandler>	Y	A map of attachments to an outbound message. The key is a unique identifier for the attachment. The value is a <code>DataHandler</code> for the attachment data. Bindings describe how to carry attachments with messages.
javax.xml.ws.reference			
<code>.parameters</code>	List<Element>	Y	A list of WS Addressing reference parameters. The list MUST include all SOAP headers marked with the <code>wsa:IsReferenceParameter= true</code> attribute.
javax.xml.ws.wsdl			
<code>.description</code>	URI	N	A resolvable URI that may be used to obtain access to the WSDL for the endpoint.
<code>.service</code>	QName	N	The name of the service being invoked in the WSDL.

<code>.port</code>	QName	N	The name of the port over which the current message was received in the WSDL.
<code>.interface</code>	QName	N	The name of the port type to which the current message belongs.
<code>.operation</code>	QName	N	The name of the WSDL operation to which the current message belongs. The namespace is the target namespace of the WSDL definitions element.

Table 9.3: Standard HTTP `MessageContext` properties.

Name	Type	Mandatory	Description
javax.xml.ws.http.request			
<code>.headers</code>	Map<String,List<String>>	Y	A map of the HTTP headers for the request message. The key is the header name. The value is a list of values for that header.
<code>.method</code>	String	Y	The HTTP method for the request message.

<code>.querystring</code>	<code>String</code>	Y	The HTTP query string for the request message, or <code>null</code> if the request does not have any. If the address specified using the <code>javax.xml.ws.service.endpoint.address</code> in the <code>BindingProvider</code> contains a query string and if the <code>querystring</code> property is set by the client it will override the existing query string in the <code>javax.xml.ws.service.endpoint.address</code> property. The value of the property does not include the leading <code>` `?"</code> of the query string in it. This property is only used with HTTP binding.
<code>.pathinfo</code>	<code>String</code>	Y	Extra path information associated with the URL the client sent when it made this request. The extra path information follows the base url path but precedes the query string and will start with a <code>"/"</code> character.
<code>javax.xml.ws.http.response</code>			
<code>.headers</code>	<code>Map<String,List<String>></code>	Y	A map of the HTTP headers for the response message. The key is the header name. The value is a list of values for that header.
<code>.code</code>	<code>Integer</code>	Y	The HTTP response status code.

Table 9.4: Standard Servlet Container-Specific `MessageContext` properties.

Name	Type	Mandatory	Description
<code>javax.xml.ws.servlet</code>			

<code>.context</code>	<code>javax.servlet.ServletContext</code>	Y	The <code>ServletContext</code> object belonging to the web application that contains the endpoint.
<code>.request</code>	<code>javax.servlet.http.HttpServletRequest</code>	Y	The <code>HttpServletRequest</code> object associated with the request currently being served.
<code>.response</code>	<code>javax.servlet.http.HttpServletResponse</code>	Y	The <code>HttpServletResponse</code> object associated with the request currently being served.

10.4.2. `javax.xml.ws.handler.LogicalMessageContext`

Logical handlers (see section [Types of Handler](#)) are passed a message context of type `LogicalMessageContext` when invoked. `LogicalMessageContext` extends `MessageContext` with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of a message. A protocol binding defines what component of a message are available via a logical message context. E.g., the SOAP binding, see section [SOAP Handlers](#), defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers whereas the XML/HTTP binding described in chapter [HTTP Binding](#) defines that a logical handler can access the entire XML payload of a message.

The `getSource()` method of `LogicalMessageContext` MUST return null whenever the message doesn't contain an actual payload. A case in which this might happen is when, on the server, the endpoint implementation has thrown an exception and the protocol in use does not define a notion of payload for faults (e.g. the HTTP binding defined in chapter [HTTP Binding](#)).

10.4.3. Relationship to Application Contexts

Client side binding providers have methods to access contexts for outbound and inbound messages. As described in section [Configuration](#) these contexts are used to initialize a message context at the start of a message exchange and to obtain application scoped properties from a message context at the end of a message exchange.

As described in chapter [Service APIs](#), service endpoint implementations may require injection of a context from which they can access the message context for each inbound message and manipulate the corresponding application-scoped properties.

Handlers may manipulate the values and scope of properties within the message context as desired. E.g., a handler in a client-side SOAP binding might introduce a header into a SOAP request message to carry metadata from a property that originated in a `BindingProvider` request context; a handler in a server-side SOAP binding might add application scoped properties to the message context from the contents of a header in a request SOAP message that is then made available in the context available

(via injection) to a service endpoint implementation.

Chapter 11. SOAP Binding

This chapter describes the JAX-WS SOAP binding and its extensions to the handler framework (described in chapter [Handler Framework](#)) for SOAP message processing.

11.1. Configuration

A SOAP binding instance requires SOAP specific configuration in addition to that described in section [Configuration](#). The additional information can be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

11.1.1. Programmatic Configuration

JAX-WS defines APIs for programmatic configuration of client-side SOAP bindings. Server side bindings can be configured programmatically when using the [Endpoint](#) API (see [javax.xml.ws.Endpoint](#)), but are otherwise expected to be configured using annotations or deployment metadata.

11.1.1.1. SOAP Roles

SOAP 1.1[\[2\]](#) and SOAP 1.2[\[3\]\[4\]](#) use different terminology for the same concept: a SOAP 1.1 *actor* is equivalent to a SOAP 1.2 *role*. This specification uses the SOAP 1.2 terminology.

An ultimate SOAP receiver always plays the following roles:

Next

In SOAP 1.1, the next role is identified by the URI <http://schemas.xmlsoap.org/soap/actor/next>. In SOAP 1.2, the next role is identified by the URI <http://www.w3.org/2003/05/soap-envelope/role/next>.

Ultimate receiver

In SOAP 1.1 the ultimate receiver role is identified by omission of the [actor](#) attribute from a SOAP header. In SOAP 1.2 the ultimate receiver role is identified by the URI <http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver> or by omission of the [role](#) attribute from a SOAP header.

Conformance (SOAP required roles): An implementation of the SOAP binding MUST act in the following roles: next and ultimate receiver.

A SOAP 1.2 endpoint never plays the following role:

None

In SOAP 1.2, the none role is identified by the URI <http://www.w3.org/2003/05/soap-envelope/role/none>.

Conformance (SOAP required roles): An implementation of the SOAP binding MUST NOT act in the none role.

The `javax.xml.ws.SOAPBinding` interface is an abstraction of the JAX-WS SOAP binding. It extends `javax.xml.ws.Binding` with methods to configure additional SOAP roles played by the endpoint.

Conformance (Default role visibility): An implementation MUST include the required next and ultimate receiver roles in the `Set` returned from `SOAPBinding.getRoles`.

Conformance (Default role persistence): An implementation MUST add the required next and ultimate receiver roles to the roles configured with `SOAPBinding.setRoles`.

Conformance (None role error): An implementation MUST throw `WebServiceException` if a client attempts to configure the binding to play the none role via `SOAPBinding.setRoles`.

11.1.1.2. SOAP Handlers

The handler chain for a SOAP binding is configured as described in section [Programmatic Configuration](#). The handler chain may contain handlers of the following types:

Logical

Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler` either directly or indirectly. Logical handlers have access to the content of the SOAP body via the logical message context.

SOAP

SOAP handlers are handlers that implement `javax.xml.ws.handler.soap.SOAPHandler`.

Mime attachments specified by the `javax.xml.ws.binding.attachments.inbound` and `javax.xml.ws.binding.attachments.outbound` properties defined in the `MessageContext` [[stdbindingpropstbl](#)] can be modified in logical handlers. A SOAP message with the attachments specified using the properties is generated before invoking the first `SOAPHandler`. Any changes to the two properties in consideration above in the `MessageContext` after invoking the first `SOAPHandler` are ignored. The `SOAPHandler` however may change the properties in the `MessageContext`.

Use of `javax.xml.ws.binding.attachments.outbound` property in Dispatch

- When using `Dispatch` in SOAP / HTTP binding in payload mode, attachments specified using the `javax.xml.ws.binding.attachments.outbound` property will be included as mime attachments in the message.
- When using `Dispatch` in SOAP / HTTP binding in message mode, the `javax.xml.ws.binding.attachments.outbound` property will be ignored as the message type already provides a way to specify attachments.

Conformance (Incompatible handlers): An implementation MUST throw `WebServiceException` when, at the time a binding provider is created, the handler chain returned by the configured `HandlerResolver` contains an incompatible handler.

Conformance (Incompatible handlers): Implementations MUST throw a `WebServiceException` when

attempting to configure an incompatible handler using `Binding.setHandlerChain`.

Conformance (Logical handler access): An implementation MUST allow access to the contents of the SOAP body via a logical message context.

11.1.1.3. SOAP Headers

The SOAP headers understood by a handler are obtained using the `getHeaders` method of `SOAPHandler`.

11.1.2. Deployment Model

JAX-WS defines no standard deployment model for handlers. Such a model is provided by JSR 109[17] Implementing Enterprise Web Services.

11.2. Processing Model

The SOAP binding implements the general handler framework processing model described in section [Processing Model](#) but extends it to include SOAP specific processing as described in the following subsections.

11.2.1. SOAP `mustUnderstand` Processing

The SOAP protocol binding performs the following additional processing on inbound SOAP messages prior to the start of normal handler invocation processing (see section [Handler Execution](#)). Refer to the SOAP specification[2][3][4] for a normative description of the SOAP processing model. This section is not intended to supercede any requirement stated within the SOAP specification, but rather to outline how the configuration information described above is combined to satisfy the SOAP requirements:

1. Obtain the set of SOAP roles for the current binding instance. This is returned by `SOAPBinding.getRoles`.
2. Obtain the set of `Handler`'s deployed on the current binding instance. This is obtained via `Binding.getHandlerChain`.
3. Identify the set of header qualified names (QNames) that the binding instance understands. This is the set of all header `QName`'s that satisfy at least one of the following conditions:
 1. that are mapped to method parameters in the service endpoint interface;
 2. are members of `SOAPHandler.getHeaders()` for each `SOAPHandler` in the set obtained in step [\[muhandinfoiset\]](#);
 3. are directly supported by the binding instance.
4. [\[mutargettedheaders\]](#) Identify the set of must understand headers in the inbound message that are targeted at this node. This is the set of all headers with a `mustUnderstand` attribute whose value is `1` or `true` and an `actor` or `role` attribute whose value is in the set obtained in step [\[muhandroleset\]](#).
5. For each header in the set obtained in step [\[mutargettedheaders\]](#), the header is understood if its QName is in the set identified in step [\[muunderstoodheaders\]](#).

6. If every header in the set obtained in step [\[mutargettedheaders\]](#) is understood, then the node understands how to process the message. Otherwise the node does not understand how to process the message.
7. If the node does not understand how to process the message, then neither handlers nor the endpoint are invoked and instead the binding generates a SOAP must understand exception. Subsequent actions depend on whether the message exchange pattern (MEP) in use requires a response to the message currently being processed or not:

Response

The message direction is reversed and the binding dispatches the SOAP must understand exception (see section [Exception Handling](#)).

No response

The binding dispatches the SOAP must understand exception (see section [Exception Handling](#)).

11.2.2. Exception Handling

The following subsections describe SOAP specific requirements for handling exceptions thrown by handlers and service endpoint implementations.

11.2.2.1. Handler Exceptions

A binding is responsible for catching runtime exceptions thrown by handlers and following the processing model described in section [Handler Execution](#). A binding is responsible for converting the exception to a fault message subject to further handler processing if the following criteria are met:

1. A handler throws a [ProtocolException](#) from `handleMessage`
2. The MEP in use includes a response to the message being processed
3. The current message is not already a fault message (the handler might have undertaken the work prior to throwing the exception).

If the above criteria are met then the exception is converted to a SOAP fault message as follows:

- If the exception is an instance of [SOAPFaultException](#) then the fields of the contained SAAJ [SOAPFault](#) are serialized to a new SOAP fault message, see section [Mapping Exceptions to SOAP Faults](#). The current message is replaced by the new SOAP fault message.
- If the exception is of any other type then a new SOAP fault message is created to reflect a server class of error for SOAP 1.1[\[2\]](#) or a receiver class of error for SOAP 1.2[\[3\]](#).
- Handler processing is resumed as described in section [Handler Execution](#).

If the criteria for converting the exception to a fault message subject to further handler processing are not met then the exception is handled as follows depending on the current message direction:

Outbound

A new SOAP fault message is created to reflect a server class of error for SOAP 1.1[2] or a receiver class of error for SOAP 1.2[3] and the message is dispatched.

Inbound

The exception is passed to the binding provider.

11.2.2.2. Service Endpoint Exceptions

Service endpoints can throw service specific exceptions or runtime exceptions. In both cases they can provide protocol specific information using the cause mechanism, see section [Protocol Specific Exception Handling](#).

A server side implementation of the SOAP binding is responsible for catching exceptions thrown by a service endpoint implementation and, if the message exchange pattern in use includes a response to the message that caused the exception, converting such exceptions to SOAP fault messages and invoking the `handleFault` method on handlers for the fault message as described in section [Handler Execution](#).

Section [Mapping Exceptions to SOAP Faults](#) describes the rules for mapping an exception to a SOAP fault.

11.2.2.3. Mapping Exceptions to SOAP Faults

When mapping an exception to a SOAP fault, the fields of the fault message are populated according to the following rules of precedence:

- **faultcode** (Subcode in SOAP 1.2, Code set to `env:Receiver`)
 1. `SOAPFaultException.getFault().getFaultCodeAsQName()`
 2. `env:Server` (Subcode omitted for SOAP 1.2).
- **faultstring** (Reason/Text)
 1. `SOAPFaultException.getFault().getFaultString()`
 2. `Exception.getMessage()`
 3. `Exception.toString()`
- **faultactor** (Role in SOAP 1.2)
 1. `SOAPFaultException.getFault().getFaultActor()`
 2. Empty
- **detail** (Detail in SOAP 1.2)
 1. Serialized service specific exception (see `WrapperException.getFaultInfo()` in section [Fault](#))
 2. `SOAPFaultException.getFault().getDetail()`

11.3. SOAP Message Context

SOAP handlers are passed a `SOAPMessageContext` when invoked. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.

11.4. SOAP Transport and Transfer Bindings

SOAP[2][4] can be bound to multiple transport or transfer protocols. This section describes requirements pertaining to the supported protocols for use with SOAP.

11.4.1. HTTP

The SOAP 1.1 HTTP binding is identified by the URL <http://schemas.xmlsoap.org/wsdl/soap/http>, which is also the value of the constant `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`.

Conformance (SOAP 1.1 HTTP Binding Support): An implementation MUST support the HTTP binding of SOAP 1.1[2] and SOAP With Attachments[38] as clarified by the WS-I Basic Profile[20], WS-I Simple SOAP Binding Profile[32] and WS-I Attachment Profile[33].

The SOAP 1.2 HTTP binding is identified by the URL <http://www.w3.org/2003/05/soap/bindings/HTTP/>, which is also the value of the constant `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING`.

Conformance (SOAP 1.2 HTTP Binding Support): An implementation MUST support the HTTP binding of SOAP 1.2[4].

11.4.1.1. MTOM

Conformance (SOAP MTOM Support): An implementation MUST support MTOM[30] [12: JAX-WS inherits the JAXB support for the SOAP MTOM[30]/XOP[31] mechanism for optimizing transmission of binary data types, see section [Types](#).].

`SOAPBinding` defines a property (in the JavaBeans sense) called `MTOMEnabled` that can be used to control the use of MTOM. The `getMTOMEnabled` method is used to query the current value of the property. The `setMTOMEnabled` method is used to change the value of the property so as to enable or disable the use of MTOM.

Conformance (Semantics of MTOM enabled): When MTOM is enabled, a receiver MUST accept both non-optimized and optimized messages, and a sender MAY send an optimized message, non-optimized messages being also acceptable.

The heuristics used by a sender to determine whether to use optimization or not are implementation-specific.

Conformance (MTOM support): Predefined `SOAPBinding` instances MUST support enabling/disabling MTOM support using the `setMTOMEnabled` method.

Conformance (SOAP bindings with MTOM disabled): The bindings corresponding to the following IDs:

- `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`
- `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING`

MUST have MTOM disabled by default.

For convenience, this specification defines two additional binding identifiers for SOAP 1.1 and SOAP 1.2 over HTTP with MTOM enabled.

The URL of the former is <http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true> and its predefined constant `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING`.

The URL of the latter is <http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true> and its predefined constant `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING`.

Conformance (SOAP bindings with MTOM enabled): The bindings corresponding to the following IDs:

- `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING`
- `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING`

MUST have MTOM enabled by default.

Conformance (MTOM on Other SOAP Bindings): Other bindings that extend `SOAPBinding` MAY NOT support changing the value of the `MTOMEnabled` property. In this case, if an application attempts to change its value, an implementation MUST throw a `WebServiceException`.

11.4.1.2. One-way Operations

HTTP interactions are request-response in nature. When using HTTP as the transfer protocol for a one-way SOAP message, implementations wait for the HTTP response even though there is no SOAP message in the HTTP response entity body.

Conformance (One-way operations): When invoking one-way operations, an implementation of the SOAP/HTTP binding MUST block until the HTTP response is received or an error occurs.

Note that completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

11.4.1.3. Security

Section [Standard Properties](#) defines two standard context properties (`javax.xml.ws.security.auth.username` and `javax.xml.ws.security.auth.password`) that may be used to configure authentication information.

Conformance (HTTP basic authentication support): An implementation of the SOAP/HTTP binding MUST support HTTP basic authentication.

Conformance (Authentication properties): A client side implementation MUST support use of the the standard properties `javax.xml.ws.security.auth.username` and `javax.xml.ws.security.auth.password` to configure HTTP basic authentication.

11.4.1.4. Session Management

Section [Standard Properties](#) defines a standard context property (`javax.xml.ws.session.maintain`) that may be used to control whether a client side runtime will join a session initiated by a service.

A SOAP/HTTP binding implementation can use three HTTP mechanisms for session management:

Cookies

To initiate a session a service includes a cookie in a message sent to a client. The client stores the cookie and returns it in subsequent messages to the service.

URL rewriting

To initiate a session a service directs a client to a new URL for subsequent interactions. The new URL contains an encoded session identifier.

SSL

The SSL session ID is used to track a session.

R1120 in WS-I Basic Profile 1.1[\[20\]](#) allows a service to use HTTP cookies. However, R1121 recommends that a service should not rely on use of cookies for state management.

[URL rewriting support] An implementation MUST support use of HTTP URL rewriting for state management.

[Cookie support] An implementation SHOULD support use of HTTP cookies for state management.

[SSL session support] An implementation MAY support use of SSL session based state management.

11.4.1.5. Addressing

If addressing is enabled, implementations are required to follow WS-Addressing[\[26\]](#)[\[37\]](#)[\[27\]](#) protocols.

Conformance (SOAP Addressing Support): An implementation MUST support WS-Addressing 1.0 - SOAP Binding[\[37\]](#).

– Conformance (`wsa:Action` value): `wsa:Action` value MUST be got from `@Action` annotation elements in SEI, if present. But if a client sets a `BindingProvider.SOAPACTION_URI_PROPERTY` property then that MUST be used for `wsa:Action` header.

If a receiver receives messages with the WS-Addressing headers that are non-conformant as per WS-Addressing 1.0- SOAP Binding[\[37\]](#), then appropriate addressing pre-defined faults must be generated.

A JAX-WS application may send `wsa:replyTo` or `wsa:FaultTo` addressing header to receive non-

anonymous responses at a different address other than the transport back channel. When the application receives a response at a different address, there is no standard way to communicate the response with the JAX-WS client runtime. Hence, there are no requirements on a JAX-WS client runtime to bind non-anonymous responses. A JAX-WS client runtime may start an endpoint to receive a non-anonymous response and may use the response to bind to the java parameters and return type. However, it is not required to do so.

Chapter 12. HTTP Binding

This chapter describes the JAX-WS XML/HTTP binding. The JAX-WS XML/HTTP binding provides raw XML over HTTP messaging capabilities as used in many Web services today.

12.1. Configuration

The XML/HTTP binding is identified by the URL <http://www.w3.org/2004/08/wsdl/http>, which is also the value of the constant `javax.xml.ws.http.HTTPBinding.HTTP_BINDING`.

Conformance (XML/HTTP Binding Support): An implementation MUST support the XML/HTTP binding.

An XML/HTTP binding instance allows HTTP-specific configuration in addition to that described in section [Configuration](#). The additional information can be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

12.1.1. Programmatic Configuration

JAX-WS only defines APIs for programmatic configuration of client side XML/HTTP bindings – server side bindings are expected to be configured using deployment metadata.

12.1.1.1. HTTP Handlers

The handler chain for an XML/HTTP binding is configured as described in section [Programmatic Configuration](#). The handler chain may contain handlers of the following types:

Logical

Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler` either directly or indirectly. Logical handlers have access to the entire XML message via the logical message context.

Use of `javax.xml.ws.binding.attachments.outbound` property in Dispatch

- When using `Dispatch` in XML / HTTP binding in payload mode, attachments specified using the `javax.xml.ws.binding.attachments.outbound` property will be included as mime attachments to the message.
- When using `Dispatch` in XML / HTTP binding in message mode, the `javax.xml.ws.binding.attachments.outbound` property will be ignored. `Dispatch` of type `DataSource` should be used to send mime attachments for the XML / HTTP binding in message mode.

Conformance (Incompatible handlers): An implementation MUST throw `WebServiceException` when, at the time a binding provider is created, the handler chain returned by the configured `HandlerResolver` contains an incompatible handler.

Conformance (Incompatible handlers): Implementations MUST throw a `WebServiceException` when attempting to configure an incompatible handler using `Binding.setHandlerChain`.

Conformance (Logical handler access): An implementation MUST allow access to the entire XML message via a logical message context.

12.1.2. Deployment Model

JAX-WS defines no standard deployment model for handlers. Such a model is provided by JSR 109[\[17\]](#) Implementing Enterprise Web Services.

12.2. Processing Model

The XML/HTTP binding implements the general handler framework processing model described in section [Processing Model](#).

12.2.1. Exception Handling

The following subsections describe HTTP specific requirements for handling exceptions thrown by handlers and service endpoint implementations.

12.2.1.1. Handler Exceptions

A binding is responsible for catching runtime exceptions thrown by handlers and following the processing model described in section [Handler Execution](#). A binding is responsible for converting the exception to a fault message subject to further handler processing if the following criteria are met:

1. A handler throws a `ProtocolException` from `handleMessage`
2. The MEP in use includes a response to the message being processed
3. The current message is not already a fault message (the handler might have undertaken the work prior to throwing the exception).

If the above criteria are met then the exception is converted to a HTTP response message as follows:

- If the exception is an instance of `HTTPException` then the HTTP response code is set according to the value of the `statusCode` property. Any current XML message content is removed.
- If the exception is of any other type then the HTTP status code is set to 500 to reflect a server class of error and any current XML message content is removed.
- Handler processing is resumed as described in section [Handler Execution](#).

If the criteria for converting the exception to a fault message subject to further handler processing are not met then the exception is handled as follows depending on the current message direction:

Outbound

The HTTP status code is set to 500 to reflect a server class of error, any current XML message content is removed and the message is dispatched.

Inbound

The exception is passed to the binding provider.

12.2.1.2. Service Endpoint Exceptions

Service endpoints can throw service specific exceptions or runtime exceptions. In both cases they can provide protocol specific information using the cause mechanism, see section [Protocol Specific Exception Handling](#).

A server side implementation of the XML/HTTP binding is responsible for catching exceptions thrown by a service endpoint implementation and, if the message exchange pattern in use includes a response to the message that caused the exception, converting such exceptions to HTTP response messages and invoking the `handleFault` method on handlers for the response message as described in section [Handler Execution](#).

Section [Mapping Exceptions to a HTTP Status Code](#) describes the rules for mapping an exception to a HTTP status code.

12.2.1.3. Mapping Exceptions to a HTTP Status Code

When mapping an exception to a HTTP status code, the status code of the HTTP fault message is populated according to the following rules of precedence:

1. ``HTTPException.getStatusCode()`` [13: If the exception is a `HTTPException` or has a cause that is a `HTTPException` .]
2. `500`.

12.3. HTTP Support

12.3.1. One-way Operations

HTTP interactions are request-response in nature. When used for one-way messages, implementations wait for the HTTP response even though there is no XML message in the HTTP response entity body.

Conformance (One-way operations): When invoking one-way operations, an implementation of the XML/HTTP binding MUST block until the HTTP response is received or an error occurs.

Note that completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

12.3.2. Security

Section [Standard Properties](#) defines two standard context properties

(`javax.xml.ws.security.auth.username` and `javax.xml.ws.security.auth.password`) that may be used to configure authentication information.

Conformance (HTTP basic authentication support): An implementation of the XML/HTTP binding MUST support HTTP basic authentication.

Conformance (Authentication properties): A client side implementation MUST support use of the the standard properties `javax.xml.ws.security.auth.username` and `javax.xml.ws.security.auth.password` to configure HTTP basic authentication.

12.3.3. Session Management

Section [Standard Properties](#) defines a standard context property (`javax.xml.ws.session.maintain`) that may be used to control whether a client side runtime will join a session initiated by a service.

A XML/HTTP binding implementation can use three HTTP mechanisms for session management:

Cookies

To initiate a session a service includes a cookie in a message sent to a client. The client stores the cookie and returns it in subsequent messages to the service.

URL rewriting

To initiate a session a service directs a client to a new URL for subsequent interactions. The new URL contains an encoded session identifier.

SSL

The SSL session ID is used to track a session.

Conformance (URL rewriting support) An implementation MUST support use of HTTP URL rewriting for state management.

Conformance (Cookie support): An implementation SHOULD support use of HTTP cookies for state management.

Conformance (SSL session support): An implementation MAY support use of SSL session based state management.

Appendix A: Conformance Requirements

Appendix B: Change Log

2.3 Changes

- Changed the definition of the discovery process of `javax.xml.ws.spi.Provider` implementation (section [Configuration](#))
- Added `@Repeatable` to `@WebServiceClientRef`
- Added missing javadoc
- Added `@since` to javadoc
- Updated references from J2SE to Java SE

2.2 Changes

- Added `@Action` generation requirements in SEI (section [Message and Part](#))
- Updated `wsa:EndpointReference` mapping (section [W3CEndpointReference](#))
- Updated that the use of metadata in `mime:content` is disabled by default (section `mime:content`)
- Updated generated service constructor requirements (section [Service and Port, Static case](#))
- Updated that static or final methods cannot be web methods (section [Class](#))
- Added `wsam:Action` algorithm and requirements (section [Method](#), [wsam:Action Computation Algorithm](#))
- Clarified the namespace of the generated JAXB beans (section [Method Parameters and Return Type](#))
- Added that packaging wrapper beans optional (section [Document Wrapped](#))
- Added requirements for wsdl:part names in doc/lit wrapper case (section [Document Wrapped](#))
- Added that `@XmlElement` can be specified on doc/lit wrapper method parameters (section [Document Wrapped](#))
- Added wsdl:message naming requirements (section [Service Specific Exception](#))
- Added `@XmlType` requirements on exception classes (section [Service Specific Exception](#))
- Clarified the fault bean's `@XmlType` requirements (section [Service Specific Exception](#))
- Added that packaging exception beans optional (section [Service Specific Exception](#))
- Added Use of Addressing requirements in WSDL (section [Service and Ports](#))
- Added service creation with web service features (section [Dynamic case](#))
- Added `W3CEndpointReference` requirements (section [javax.xml.ws.BindingProvider](#), [javax.xml.ws.EndpointReference](#), [javax.xml.ws.WebServiceContext](#), [javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder](#))
- Added Use of Addressing requirements in proxies (section [Proxies](#))

- Clarified the empty payload Source for Provider (section [javax.xml.ws.Provider](#))
- Clarified oneway invocation for Provider endpoints (section [Invocation](#))
- Added features usage with Endpoint API (section [Endpoint Usage, Creating Endpoint Objects](#))
- Added HTTP SPI (section [javax.xml.ws.spi.http \(HTTP SPI\)](#))
- Clarified that fully qualified class name can be specified for generated exception, service classes customizations (section [PortType Fault Message, Service](#))

2.1 Changes

- Added JAXB 2.1 requirement (sections 1.1, 2.4, 3.6.2).
- Added wsa namespace definition (section 1.5).
- Added conformance requirement for `@XmlSeeAlso` annotation (section 2.2).
- Added conformance requirement for use of JAXB annotations (section 2.3.1).
- Added clarification that not both input and output messages must be present for wrapper style (section 2.3.1.2).
- Added section 2.4.1 `W3CEndpointReference`.
- Added `getPortName(WebServiceFeature...)` method to generated Service (section 2.7).
- Added text describing the need to use customizations to resolve some conflicts (section 3.4).
- Added conformance requirement to honor JAXB annotations (section 3.6).
- Added conformance requirement for Exceptions that are NOT service specific exceptions (section 3.7).
- Added conformance requirement for `BindingProvider.getEndpointReference` (section 4.2).
- Added new `getPort` methods on `Service` that take `WebServiceFeatures` and `EndpointReference` (section 4.2.3).
- Added text stating that `Dispatch` and `Provider` based applications MUST honor `WebServiceFeatures` (section 4.3 and 5.1).
- Added sections `javax.xml.ws.EndpointReference`, `javax.xml.ws.EndpointReference` and `javax.xml.ws.EndpointReference javax.xml.ws.EndpointReference`.
- Added section 5.4 on `javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder`.
- Modified description of `createEndpoint` method to state to cover case when no binding is specified (section [Creating Endpoint Objects](#)).
- Added section [EndpointReferences](#) EndpointReferences.
- Added section [Getting Port Objects](#) Getting Port Objects.
- Added section `javax.xml.ws.WebServiceFeature` `javax.xml.ws.WebServiceFeature`.
- Added conformance requirement for unsupported `WebServiceFeatureAnnotations` (section 7).

- Added `Action`, `FaultAction` and `WebServiceFeatureAnnotation` annotations (sections 7.12, 7.13 and 7.14).
- Added `javax.xml.ws.reference.parameters` standard message context property (table 9.2).
- Added section [Addressing](#) Addressing.

Changes since Proposed Final Draft

- Added clarification for usage of `javax.xml.ws.binding.attachments.outbound` in Dispatch
- Added clarification for usage of `null` in Dispatch (section 4.3).
- Removed requirement that the `name` element of the `WebFault` annotation be always present, since this conflicts with 3.7 (section 7.2).
- Clarified usage of generics in document wrapped case.
- Added inner class mapping requirements.
- Rephrased rules on using `WebServiceContext` so that the limitations that apply in the Java SE environment are marked as such (section 5.3).
- Added conformance requirements for `RequestWrapper` and `ResponseWrapper` annotations (section 2.3.1.2).
- Clarified order of invocation of `Handler.close` methods (section 9.3.2.3).
- Clarified requirement on additional context properties and reserved the `java.*` and `javax.*` namespaces for Java specifications (section 4.2.1.2).
- Added new binding identifiers for SOAP/HTTP bindings with MTOM enabled (section 10.4.1.1).
- Added requirement detailing the semantics of `MTOM enabled` (section 10.4.1.1).
- Renamed section 5.2.5 and added new requirements around generation of the contract for an endpoint (section 5.2.5).
- Fixed example in figure 3.4 and added requirement on `XmlType` annotation on a generated fault bean (section 3.7).
- Removed references to WSDL 2.0 and updated goals to reflect WSDL 2.0 support will be added a future revision of the specification.
- Clarified the nullability status of various elements in the Java to WSDL binding (sections 3.6.2.1, 3.6.2.2); this included adding a new conformance requirement (section 3.6.2.3).
- Added a requirement that a class annotated with `WebServiceProvider` must not be annotated with `WebService` (section 7.7).
- Added a conformance requirement for support of the XML/HTTP binding, in analogy with the existing requirements for SOAP (section 11.1).
- Added explicit mention of the predefined binding identifiers (sections 10.4.1 and 11.1).
- Added requirements around binding identifiers for implementation-specific bindings (section 6.1).

- Adding a requirement on dealing with exceptions thrown during handler processing (section 4.2.4).
- Removed the `javax.xml.ws.servlet.session` message context property (section 9.4.1.1).
- Fixed erroneous reference to a `generated service interface` in section 7.9 (the correct terminology is `generated service class`).
- Added `javax.xml.ws.WebServiceRefs` annotation (section 7.10).
- Added clarifications for XML / HTTP binding.
- Corrected signature for `Endpoint.create` to use `String` for `bindingId`.

Changes since Public Draft

- Changed endpoint publishing so that endpoints cannot be stopped and published again multiple times (section 5.2.2).
- Clarified that request and response beans do not contain properties corresponding to header parameters (section 3.6.2.1).
- Clarified that criteria for bare style take only parts bound to the body into account (section 3.6.2.2).
- Add a `create(Object implementor)` to `Endpoint` to create an `Endpoint`.
- Clarified the use of `INOUT` param with two different MIME bindings in the input and output messages.
- Use of `WebParam` and `WebResult` `partName`.
- Replaced the `init/destroy` methods of handlers with the `PostConstruct` and `PreDestroy` annotations from JSR-250 (section 9.3.1).
- Replaced the `BeginService/EndService` annotations with `PostConstruct` and `PreDestroy` from JSR-250 in endpoint implementors (section 5.2.1).
- Added `WebParam.header` `WebResult.header` usage (section 3.6) and updated WSDL SOAP HTTP Binding section (3.9.2).
- Removed requirements to support additional SOAP headers mapping.
- Added support for `DataSource` as a message format for `Provider` and clarified the requirement for the other supported types (section 5.1). Same thing for `Dispatch` (section 4.3).
- Clarified that `LogicalMessageContext.getSource()` may return `null` when there is no payload associated with the message (section 9.4.2).
- Clarified that parts bound to `mime:content` are treated as unlisted from the point of view of applying the wrapper style rules (section 2.6.3).
- Removed the `ParameterIndex` annotation (chapters 3 and 7).
- Clarified naming rules for generated wrapper elements and their type (section 3.6.2.1).
- Clarified that holders should never be used for the return type of a method (section 2.3.3).
- Added effect of the `BindingType` annotation on the generated WSDL service (sections 3.8 and 3.10).

- Added condition that the wrapper elements be non-nillable to wrapper style (section 2.3.1.2).
- Clarified use of targetNamespace from WebService in an implementation class and an SEI based on 181 changes.
- Updated the usage of WebMethod exclude element from Java to WSDL mapping.
- Changed the algorithm for the default target namespace from java to WSDL (section 3.2).
- Added requirement that a provider's constructor be public (section 5.1).
- Fixed some inconsistencies caused by the removal of RemoteException (e.g. in section 4.2.4).
- Added service delegate requirements to chapter 4.
- Added zero-argument public constructor requirement to the implementation-specific Provider SPI class (section 6.2).
- Updated use of SOAPBinding on a per method basis in the document style case and removed editor's note about SOAPBinding not being allowed on methods (section 2.3.1 and 3.6.2) .
- Added portName element to @WebServiceProvider annotation.
- Added requirement on correctness of annotation to the beginning of chapter 7.
- Added requirement for conformance to the JAX-WS profile in JSR-181 (section 7.11).
- Clarified invocation of Handler.destroy (section 9.3.1).
- Added use of HandlerChain annotation (section 9.2.1.3).
- Updated 181 annotations (section 7.11).
- Added catalog facility (section 4.2.5) and clarified that it is required to be used when processing endpoint metadata at publishing time (section 5.2.5) and annotations (chapter 7).
- Added WebServiceRef annotation (section 7.10).
- Added restrictions on metadata at the time an endpoint is published (section 5.2.7).
- Replaced HandlerRegistry with HandlerResolver (sections 4.2.1, 9.2.1.1, 10.1.1.2, 11.1.1.1). Fixed handler ordering section accordingly (section 9.2.1.2).
- Clarified that endpoint properties override the values defined using the WebServiceProvider annotation (section 5.2.8).
- Removed mapping of headerfaults (sections 2.6.2.2 and 8.7.6).
- Split standard message context properties into multiple tables and added servlet-specific properties (section 9.4.1.1).
- Added WebServiceContext (section 5.3); refactored message context section in chapter 5 so that it applies to all kinds of endpoints.
- Added WebServicePermission (section 5.2.5).
- Added conformance requirement for one-way operations (section 6.2.2).
- Added BindingType annotation (section 7.9).
- Added requirement the provider endpoint implementation class carry a WebServiceProvider

annotation (section 5.1).

- Fixed RequestWrapper and ResponseWrapper description to use that they can be applied to the methods of an SEI (sections 7.4 and 7.5).
- Fixed package name for javax.xml.ws.Provider and updated section with WebServiceProvider annotation (section 5.1).
- Added WebServiceProvider annotation in `javax.xml.ws` package (section 7.8).
- Changed Factory pattern to use javax.xml.ws.spi.Provider
- Removed javax.xml.ws.EndpointFactory (section 5.2).
- Removed javax.xml.ws.Servicefactory (section 4.1).
- Removed definition of message-level security annotations (section 7.1), their use (sections 4.2.2 and 6.1.1) and the corresponding message context property (in section 9.4).
- Removed WSDL 2.0 mapping (appendices A and B).

Changes Since Early Draft 3

- Added requirements on mapping @WebService-annotated Java classes to WSDL.
- Removed references to the RMI classes that JAX-RPC 1.1 used to denote remoteness, since their role is now taken by annotations: `java.rmi.Remote` and `java.rmi.RemoteException`.
- Added `javax.xml.ws.Endpoint` on the new Endpoint API.
- Added the following new annotation types: `@RequestWrapper`, `@ResponseWrapper`, `@WebServiceImpl`, `@WebEndpoint`.
- Added the `createService(Class serviceInterface)` method to ServiceFactory.
- Renamed the `Service.createPort` method to `Service.addPort`.
- Added MTOMEnabled property to SOAPBinding.
- Removed the HTTP method getter/setter from HTTPBinding and replaced them with a new message context property called `javax.xml.ws.http.request.method`.
- In section [SOAP mustUnderstand Processing](#) clarified that SOAP headers directly supported by a binding must be treated as understood when processing mustUnderstand attributes.
- Added `getStackTrace` to the list of getters defined on `java.lang.Throwable` with must not be mapped to fault bean properties.
- In section [Standard Properties](#), removed the requirement that an exception be thrown if the application attempts to set an unknown or unsupported property on a binding provider, since there are no stub-specific properties any more, only those in the request context.
- Changed the client API chapter to reflect the annotation-based runtime. In particular, the distinction between generated stubs and dynamic proxies disappeared, and the spec now simply talks about proxies.
- Changed JAX-RPC to JAX-WS, `javax.xml.rpc.xxx` to `javax.xml.ws.xxx`. Reflected resulting changes

made to APIs.

- Added new context properties to provide access to HTTP headers and status code.
- Added new XML/HTTP Binding, see chapter [HTTP Binding](#).

Changes Since Early Draft 2

- Renamed `element''` attribute of the `jaxws:parameter` annotation to `childParameterName` for clarity, see sections [PortType Operation](#) and [Binding Operation](#).
- Added `javax.xml.ws.ServiceMode` annotation type, see section [javax.xml.ws.ServiceMode](#).
- Fixed example of external binding file to use a schema annotation, see section [External Binding File](#).
- Modified `Dispatch` so it can be used with multiple message types and either message payloads or entire messages, see section [javax.xml.ws.Dispatch](#).
- Modified `Provider` so it can be used with multiple message types and either message payloads or entire messages, see section [javax.xml.ws.Provider](#).
- Added new annotation for generated exceptions, see section [javax.xml.ws.WebFault](#).
- Added default Java package name to WSDL `targetNamespace` mapping algorithm, see section [Package](#).
- Added ordering to properties in request and response beans for `doc/lit/wrapped`, see section [Document Wrapped](#).
- Clarified that SEI method should throw JAX-RPC exception with a cause of any runtime exception thrown during local processing, see section [Exceptions](#).
- Removed requirement that SEIs MUST NOT have constants, see section [Interface](#).
- Updated document bare mapping to clarify that `@WebParam` and `@WebResult` can be used to customize the generated global element names, see section [Document Bare](#).

Changes Since Early Draft 1

- Added chapter [Service APIs](#) Service APIs.
- Added chapter [\[wsdl202java\]](#) WSDL 2.0 to Java Mapping.
- Added chapter [\[j2wsdl20chap\]](#) Java to WSDL 2.0 Mapping.
- Added mapping from Java to `wsdl:service` and `wsdl:port`, see sections [Interface](#), [Interface](#) and [Service and Ports](#).
- Fixed section [Types](#) to allow use of JAXB interface based mapping.
- Added support for document/literal/bare mapping in Java to WSDL mapping, see section [Method Parameters and Return Type](#).
- Added conformance requirement to describe the expected behaviour when two or more faults

refer to the same global element, see section [Fault](#).

- Added resolution to issue regarding binding of duplicate headers, see section [Header Binding Extension](#).
- Added use of JAXB ns URI to Java package name mapping, see section [Definitions](#).
- Added use of JAXB package name to ns URI mapping, see section [Package](#).
- Introduced new typographic convention to clearly mark non-normative notes.
- Removed references to J2EE and JNDI usage from ServiceFactory description, see section [\[svcfactusage\]](#).
- Clarified relationship between TypeMappingRegistry and JAXB.
- Emphasized control nature of context properties, added lifecycle subsection.
- Clarified fixed binding requirement for proxies.
- Added section for SOAP protocol bindings [SOAP Transport and Transfer Bindings](#). The HTTP subsection of this now contains much of the material from the JAX-RPC 1.1 Runtime Services chapter.
- Clarified that async methods are added to the regular sync SEI when async mapping is enabled rather than to a separate async-only SEI, see section [Asynchrony](#).
- Added support for WSDL MIME binding, see section [MIME Binding](#).
- Clarified that fault mapping should only generate a single exception for each equivalent set of faults, see section [Fault](#).
- Added property for message attachments.
- Removed element references to anonymous type as valid for wrapper style mapping (this doesn't prevent substitution as originally thought), see section [Wrapper Style](#).
- Removed implementation specific methods from generated service interfaces, see section [Service and Port](#).
- Clarified behaviour under fault condition for asynchronous operation mapping, see section [Faults](#).
- Clarified that additional parts mapped using soapbind:header cannot be mapped to a method return type, see section [Parameter Order and Return Type](#).
- Added new section to clarify mapping from exception to SOAP fault, see [Mapping Exceptions to SOAP Faults](#).
- Clarified meaning of *other* in the handler processing section, see [Handler Execution](#).
- Added a section to clarify Stub use of RemoteException and JAXRPCException, see [Exceptions](#).
- Added new Core API chapter and rearranged sections into Core, Client and Server API chapters.
- Changes for context refactoring, removed message context properties that previously held request/response contexts on client side, added description of rules for moving between jaxws context and message context boundaries.
- Removed requirement for Response.get to throw JAXRPCException, now throws standard

`java.util.concurrent.ExecutionException` instead.

- Added security API information, see sections [\[servicesecurityconfig\]](#) and [\[bindingmsgsec\]](#).
- Clarified SOAP `mustUnderstand` processing, see section [SOAP mustUnderstand Processing](#). Made it clear that the handler rather than the `HandlerInfo` is authoritative wrt which protocol elements (e.g. SOAP headers) it processes.
- Updated exception mapping for Java to WSDL since JAXB does not envision mapping exception classes directly, see section [Service Specific Exception](#).

Bibliography

- [bib1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Recommendation, W3C, October 2000. See <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [bib2] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. Note, W3C, May 2000. See <http://www.w3.org/TR/SOAP/>.
- [bib3] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. Recommendation, W3C, June 2003. See <http://www.w3.org/TR/2003/REC-soap12-part1-20030624>.
- [bib4] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 2: Adjuncts. Recommendation, W3C, June 2003. See <http://www.w3.org/TR/2003/REC-soap12-part2-20030624>.
- [bib5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Note, W3C, March 2001. See <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [bib6] Rahul Sharma. The Java API for XML Based RPC (JAX-RPC) 1.0. JSR, JCP, June 2002. See <http://jcp.org/en/jsr/detail?id=101>.
- [bib7] Roberto Chinnici. The Java API for XML Based RPC (JAX-RPC) 1.1. Maintenance JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=101>.
- [bib8] Keith Ballinger, David Ehnebuske, Martin Gudgin, Mark Nottingham, and Prasad Yendluri. Basic Profile Version 1.0. Final Material, WS-I, April 2004. See <http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html>.
- [bib9] Joseph Fialli and Sekhar Vajjhala. The Java Architecture for XML Binding (JAXB). JSR, JCP, January 2003. See <http://jcp.org/en/jsr/detail?id=31>.
- [bib10] Joseph Fialli and Sekhar Vajjhala. The Java Architecture for XML Binding (JAXB) 2.0. JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=222>.
- [bib11] Kohsuke Kawaguchi. The Java Architecture for XML Binding (JAXB) 2.1. JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=222>.
- [bib12] Kohsuke Kawaguchi. The Java Architecture for XML Binding (JAXB) 2.2. JSR, JCP, August 2009. See <http://jcp.org/en/jsr/detail?id=222>.
- [bib13] Kohsuke Kawaguchi. The Java Architecture for XML Binding (JAXB) 2.3. JSR, JCP, July 2017. See <http://jcp.org/en/jsr/detail?id=222>.
- [bib14] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Working Draft, W3C, August 2004. See <http://www.w3.org/TR/2004/WD-wsdl20-20040803>.
- [bib15] Joshua Bloch. A Metadata Facility for the Java Programming Language. JSR, JCP, August

2003. See <http://jcp.org/en/jsr/detail?id=175>.
- [bib16] Jim Trezzo. Web Services Metadata for the Java Platform. JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=181>.
 - [bib17] Jim Knutson and Heather Kreger. Web Services for J2EE. JSR, JCP, September 2002. See <http://jcp.org/en/jsr/detail?id=109>.
 - [bib18] Nataraj Nagaratnam. Web Services Message Security APIs. JSR, JCP, April 2002. See <http://jcp.org/en/jsr/detail?id=183>.
 - [bib19] Farrukh Najmi. Java API for XML Registries 1.0 (JAXR). JSR, JCP, June 2002. See <http://www.jcp.org/en/jsr/detail?id=93>.
 - [bib20] Keith Ballinger, David Ehnebuske, Chris Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham, Jorgen Thelin, and Prasad Yendluri. Basic Profile Version 1.1. Final Material, WS-I, August 2004. See <http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html>.
 - [bib21] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. RFC, IETF, March 1997. See <http://www.ietf.org/rfc/rfc2396.txt>.
 - [bib22] S. Bradner. RFC 2119: Keywords for use in RFCs to Indicate Requirement Levels. RFC, IETF, March 1997. See <http://www.ietf.org/rfc/rfc2119.txt>.
 - [bib23] John Cowan and Richard Tobin. XML Information Set. Recommendation, W3C, October 2001. See <http://www.w3.org/TR/2001/REC-xml-infoset-20011024/>.
 - [bib24] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Recommendation, W3C, May 2001. See <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>.
 - [bib25] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Recommendation, W3C, May +2001. See <http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/>.
 - [bib26] Tony Rogers Marting Gudgin, Marc Hadley. Web services addressing 1.0 - core. Recommendation, W3C, May 2006. See <http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/>.
 - [bib27] Marting Gudgin, Marc Hadley, Tony Rogers, and Umit Yalcinalp. Web Services Addressing 1.0 - Metadata. Recommendation, W3C, September 2007. See <http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904>.
 - [bib28] Asir S Vedomuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubetz, and Umit Yalcinalp. Web Services Services Policy 1.5 - Framework. Recommendation, W3C, September 2007. See <http://www.w3.org/TR/2007/REC-ws-policy-20070904>.
 - [bib29] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification - second edition. Book, Sun Microsystems, Inc, 2000. [http://java.sun.com/docs/books/jls/second edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second%20edition/html/j.title.doc.html).
 - [bib30] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Herve Ruellan. SOAP Message Transmission Optimization Mechanism. Recommendation, W3C, January 2005. <http://www.w3.org/TR/soap12-mtom/>.
 - [bib31] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Herve Ruellan. XML-binary

Optimized Packaging. Recommendation, W3C, January 2005. <http://www.w3.org/TR/xop10/>.

- [bib32] Mark Nottingham. Simple SOAP Binding Profile Version 1.0. Working Group Draft, WS-I, August +2004. See <http://www.ws-i.org/Profiles/SimpleSoapBindingProfile-1.0-2004-08-24.html>.
- [bib33] Chris Ferris, Anish Karmarkar, and Canyang Kevin Liu. Attachments Profile Version 1.0. Final Material, WS-I, August 2004. See <http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html>.
- [bib34] Norm Walsh. XML Catalogs 1.1. OASIS Committee Specification, OASIS, July 2005. See <http://www.oasis-open.org/committees/download.php/14041/xml-catalogs.html>.
- [bib35] Rajiv Mordani. Common Annotations for the Java Platform. JSR, JCP, July 2005. See <http://jcp.org/en/jsr/detail?id=250>.
- [bib36] Bill Shannon. Java Platform Enterprise Edition 5 Specification. JSR, JCP, August 2005. See <http://jcp.org/en/jsr/detail?id=244>.
- [bib37] Tony Rogers Marting Gudgin, Marc Hadley. Web services addressing 1.0 - soap binding. Recommendation, W3C, May 2006. See <http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509/>.
- [bib38] John Barton, Satish Thatte, and Henrik Frystyk Nielsen. SOAP Messages With Attachments. Note, W3C, December 2000. <http://www.w3.org/TR/SOAP-attachments>.