The Java API for XML Web Services (JAX-WS) 2.0

Candidate Final Draft
March 7, 2006

Editors:
Roberto Chinnici
Marc Hadley
Rajiv Mordani

Comments to: jsr224-spec-comments@sun.com

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 USA

Specification: JSR-000224 - Java™API for XML Web Services v. 2.0 (“Specification”)

Status: Pre-FCS, Post-Proposed Final Draft

Release: TBD

RESTRICTED RIGHTS LEGEND: If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Specification and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT: You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS: Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

(Sun.pre-FCS.Spec.license.11.14.2003)

Document Status

This section describes the status of this document at the time of its publication. Other documents may supersede this document; the latest revision of this document can be found on the JSR 224 homepage at `http://www.jcp.org/en/jsr/detail?id=224`. This is the Proposed Final Draft of JSR 224 (JAX-WS 2.0). It has been produced by the JSR 224 expert group. Comments on this document are welcome, send them to `jsr224-spec-comments@sun.com`.

Contents

1 Introduction **1**

1.1 Goals . . . 1

1.2 Non-Goals . . . 3

1.3 Requirements . . . 3

1.3.1 Relationship To JAXB . . . 3

1.3.2 Standardized WSDL Mapping . . . 3

1.3.3 Customizable WSDL Mapping . . . 4

1.3.4 Standardized Protocol Bindings . . . 4

1.3.5 Standardized Transport Bindings . . . 4

1.3.6 Standardized Handler Framework . . . 4

1.3.7 Versioning and Evolution . . . 5

1.3.8 Standardized Synchronous and Asynchronous Invocation . . . 5

1.3.9 Session Management . . . 5

1.4 Use Cases . . . 5

1.4.1 Handler Framework . . . 5

1.5 Conventions . . . 6

1.6 Expert Group Members . . . 7

1.7 Acknowledgements . . . 7

2 WSDL 1.1 to Java Mapping **9**

2.1 Definitions . . . 9

2.1.1 Extensibility . . . 10

2.2 Port Type . . . 10

2.3 Operation . . . 10

2.3.1 Message and Part . . . 11

2.3.2 Parameter Order and Return Type . . . 15

2.3.3 Holder Class . . . 15

2.3.4 Asynchrony . . . 16
2.4 Types . . . 20
2.5 Fault . . . 21
2.5.1 Example . . . 21
2.6 Binding . . . 23
2.6.1 General Considerations . . . 23
2.6.2 SOAP Binding . . . 23
2.6.3 MIME Binding . . . 24
2.7 Service and Port . . . 26
2.7.1 Example . . . 27
2.8 XML Names . . . 28
2.8.1 Name Collisions . . . 28

3 Java to WSDL 1.1 Mapping 29
3.1 Java Names . . . 29
3.1.1 Name Collisions . . . 29
3.2 Package . . . 29
3.3 Class . . . 30
3.4 Interface . . . 31
3.4.1 Inheritance . . . 31
3.5 Method . . . 31
3.5.1 One Way Operations . . . 32
3.6 Method Parameters and Return Type . . . 32
3.6.1 Parameter and Return Type Classification . . . 35
3.6.2 Use of JAXB . . . 36
3.7 Service Specific Exception . . . 39
3.8 Bindings . . . 40
3.8.1 Interface . . . 40
3.8.2 Method and Parameters . . . 41
3.9 Generics . . . 41
3.10 SOAP HTTP Binding . . . 43
3.10.1 Interface . . . 44
3.10.2 Method and Parameters . . . 44
3.11 Service and Ports . . . 47

4 Client APIs **49**

4.1 javax.xml.ws.Service . . . 49

4.1.1 Service Usage . . . 50

4.1.2 Provider and Service Delegate . . . 51

4.1.3 Handler Resolver . . . 51

4.1.4 Executor . . . 52

4.2 javax.xml.ws.BindingProvider . . . 52

4.2.1 Configuration . . . 52

4.2.2 Asynchronous Operations . . . 54

4.2.3 Proxies . . . 55

4.2.4 Exceptions . . . 56

4.3 javax.xml.ws.Dispatch . . . 56

4.3.1 Configuration . . . 57

4.3.2 Operation Invocation . . . 58

4.3.3 Asynchronous Response . . . 58

4.3.4 Using JAXB . . . 59

4.3.5 Examples . . . 59

4.4 Catalog Facility . . . 60

5 Service APIs **63**

5.1 javax.xml.ws.Provider . . . 63

5.1.1 Invocation . . . 64

5.1.2 Configuration . . . 64

5.1.3 Examples . . . 64

5.2 javax.xml.ws.Endpoint . . . 65

5.2.1 Endpoint Usage . . . 65

5.2.2 Publishing . . . 66

5.2.3 Publishing Permission . . . 68

5.2.4 Endpoint Metadata . . . 68

5.2.5 Determining the Contract for an Endpoint . . . 68

5.2.6 Endpoint Properties . . . 72

5.2.7 Executor . . . 72

5.3 javax.xml.ws.WebServiceContext . . . 73

5.3.1 MessageContext . . . 74

6 Core APIs **75**

6.1 javax.xml.ws.Binding . . . 75

6.2 javax.xml.ws.spi.Provider . . . 75

6.2.1 Configuration . . . 76

6.2.2 Creating Endpoint Objects . . . 76

6.2.3 Creating ServiceDelegate Objects . . . 77

6.3 javax.xml.ws.spi.ServiceDelegate . . . 77

6.4 Exceptions . . . 77

6.4.1 Protocol Specific Exception Handling . . . 77

6.4.2 One-way Operations . . . 78

7 Annotations **79**

7.1 javax.xml.ws.ServiceMode . . . 79

7.2 javax.xml.ws.WebFault . . . 80

7.3 javax.xml.ws.RequestWrapper . . . 80

7.4 javax.xml.ws.ResponseWrapper . . . 81

7.5 javax.xml.ws.WebServiceClient . . . 81

7.6 javax.xml.ws.WebEndpoint . . . 81

7.6.1 Example . . . 82

7.7 javax.xml.ws.WebServiceProvider . . . 82

7.8 javax.xml.ws.BindingType . . . 83

7.9 javax.xml.ws.WebServiceRef . . . 83

7.9.1 Example . . . 84

7.10 javax.xml.ws.WebServiceRefs . . . 85

7.10.1 Example . . . 85

7.11 Annotations Defined by JSR-181 . . . 85

7.11.1 javax.jws.WebService . . . 86

7.11.2 javax.jws.WebMethod . . . 86

7.11.3 javax.jws.OneWay . . . 86

7.11.4 javax.jws.WebParam . . . 86

7.11.5 javax.jws.WebResult . . . 86

7.11.6 javax.jws.SOAPBinding . . . 87

7.11.7 javax.jws.HandlerChain . . . 87

8 Customizations **89**

8.1 Binding Language . . . 89
8.2 Binding Declaration Container . . . 89
8.3 Embedded Binding Declarations . . . 90
8.3.1 Example . . . 90
8.4 External Binding File . . . 90
8.4.1 Example . . . 92
8.5 Using JAXB Binding Declarations . . . 92
8.6 Scoping of Bindings . . . 94
8.7 Standard Binding Declarations . . . 94
8.7.1 Definitions . . . 94
8.7.2 PortType . . . 95
8.7.3 PortType Operation . . . 96
8.7.4 PortType Fault Message . . . 97
8.7.5 Binding . . . 97
8.7.6 Binding Operation . . . 97
8.7.7 Service . . . 98
8.7.8 Port . . . 98

9 Handler Framework 101

9.1 Architecture . . . 101
9.1.1 Types of Handler . . . 101
9.1.2 Binding Responsibilities . . . 102
9.2 Configuration . . . 104
9.2.1 Programmatic Configuration . . . 104
9.2.2 Deployment Model . . . 106
9.3 Processing Model . . . 106
9.3.1 Handler Lifecycle . . . 106
9.3.2 Handler Execution . . . 107
9.3.3 Handler Implementation Considerations . . . 109
9.4 Message Context . . . 109
9.4.1 javax.xml.ws.handler.MessageContext . . . 110
9.4.2 javax.xml.ws.handler.LogicalMessageContext . . . 110
9.4.3 Relationship to Application Contexts . . . 113

10 SOAP Binding 115

10.1 Configuration . . . 115
10.1.1 Programmatic Configuration . . . 115
10.1.2 Deployment Model . . . 117
10.2 Processing Model . . . 117
10.2.1 SOAP `mustUnderstand` Processing . . . 117
10.2.2 Exception Handling . . . 118
10.3 SOAP Message Context . . . 119
10.4 SOAP Transport and Transfer Bindings . . . 119
10.4.1 HTTP . . . 119

11 HTTP Binding **123**
11.1 Configuration . . . 123
11.1.1 Programmatic Configuration . . . 123
11.1.2 Deployment Model . . . 124
11.2 Processing Model . . . 124
11.2.1 Exception Handling . . . 124
11.3 HTTP Support . . . 125
11.3.1 One-way Operations . . . 125
11.3.2 Security . . . 125
11.3.3 Session Management . . . 126

A Conformance Requirements **127**

B Change Log **133**
B.1 Changes since Proposed Final Draft . . . 133
B.2 Changes since Public Draft . . . 134
B.3 Changes Since Early Draft 3 . . . 136
B.4 Changes Since Early Draft 2 . . . 137
B.5 Changes Since Early Draft 1 . . . 138

Bibliography **141**

Chapter 1

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.

1.1 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 $\Leftrightarrow$ XML data binding. JAX-WS 2.0 will allow full use of JAXB provided facilities including binding customization and optional schema validation.

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[11] 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.

Note: *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[12] 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[13].

Implementing Enterprise Web Services (JSR 109) The JSR 109[14] defined `jaxrpc-mapping-info` deployment descriptor provides deployment time Java ⇔ WSDL mapping functionality. In conjunction with JSR 181[13], JAX-WS 2.0 will complement this mapping functionality with development time Java annotations that control Java ⇔ WSDL mapping.

Web Services Security (JSR 183) JAX-WS 2.0 will align with and complement the security APIs defined by JSR 183[15].

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 J2SE JAX-WS 2.0 will prepare JAX-WS for inclusion in a future version of J2SE. 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.

1.2 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 J2SE 5.0 JAX-WS 2.0 relies on many of the Java language features added in J2SE 5.0. It is not a goal to support JAX-WS 2.0 on Java versions prior to J2SE 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[16].

1.3 Requirements

1.3.1 Relationship To JAXB

JAX-WS describes the WSDL ⇔ Java mapping, but data binding is delegated to JAXB[10]. The specification must clearly designate where JAXB rules apply to the WSDL ⇔ 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.

1.3.2 Standardized WSDL Mapping

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

- WSDL 1.1[5] as clarified by the WS-I Basic Profile[8, 17]

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

1.3.3 Customizable WSDL Mapping

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

Java Annotations In conjunction with JAXB[10] and JSR 181[13], 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[13], 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.

1.3.4 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, 17]
- SOAP 1.2[3, 4]

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

1.3.5 Standardized Transport Bindings

The specification must describe standard bindings to the following protocols:

- HTTP/1.1[18].

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

1.3.6 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 the declarative model for handlers will be improved.

1.3.7 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.

1.3.8 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.

1.3.9 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.

1.4 Use Cases

1.4.1 Handler Framework

1.4.1.1 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.

1.4.1.2 Message Logging

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

1.4.1.3 WS-I Conformance Checking

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

1.5 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[19].

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 A.

Java code and XML fragments are formatted as shown in figure 1.1:

Figure 1.1: Example Java Code

```
1  package com.example.hello;
2
3  public class Hello {
4      public static void main(String args[]) {
5          System.out.println("Hello World");
6      }
7  }
```

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 1.1. Note that the choice of any namespace prefix is arbitrary and not semantically significant (see XML Infoset[20]).

Prefix	Namespace	Notes
env	http://www.w3.org/2003/05/soap-envelope	A normative XML Schema[21, 22] 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[21, 22] 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[21, 22]
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

Table 1.1: Prefixes and Namespaces used in this specification.

Namespace names of the general form ‘http://example.org/...’ and ‘http://example.com/...’ represent application or context-dependent URIs (see RFC 2396[18]).

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

1.6 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)
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)

1.7 Acknowledgements

Robert Bissett, Arun Gupta, Graham Hamilton, Mark Hapner, Jitendra Kotamraju, Vivek Pandey, Santiago Pericas-Geertsen, 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.

Chapter 2

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 2.6 describes binding dependent mappings.

An application MAY customize the mapping using embedded binding declarations (see section 8.3) or an external binding file (see section 8.4).

♢ *Conformance (Customization required):* Implementations MUST support customization of the WSDL 1.1 to Java mapping using the JAX-WS binding language defined in chapter 8.

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 7.11. 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.

2.1 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 8.7.1.

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[17] defined mechanisms (See R2001, R2002, and R2003) for use of WSDL and XML Schema import directives.

2.1.1 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.

2.2 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 2.1 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 2.8.

An application MAY customize this mapping using the `jaxws:class` binding declaration defined in section 8.7.2.

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

An SEI contains Java methods mapped from the `wsdl:operation` child elements of the corresponding `wsdl:portType`, see section 2.3 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.

2.3 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 2.8.

An application MAY customize this mapping using the `jaxws:method` binding declaration defined in section 8.7.3.

♢ *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[17] 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[23]) 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.

2.3.1 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 2.5). 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 2.6 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.

2.3.1.1 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 2.3.3) 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 2.8 and 2.8.1.

An application MAY customize this mapping using the `jaxws:parameter` binding declaration defined in section 8.7.3.

Section 2.3.2 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.

2.3.1.2 Wrapper Style

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

(i) The operation's input and output messages (if present) each contain only a single part

(ii) The input message part refers to a global element declaration whose localname is equal to the operation name

(iii) The output message part refers to a global element declaration

(iv) The elements referred to by the input and output message parts (henceforth referred to as *wrapper* elements) are both complex types defined using the `xsd:sequence` compositor

(v) 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 8.7.3).

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 2.4 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 2.3.3) 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 2.8 and 2.8.1.

An application MAY customize this mapping using the `jaxws:parameter` binding declaration defined in section 8.7.3.

♢ *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 2.3.2, 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.

2.3.1.3 Example

Figure 2.1 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 2.1: Wrapper and non-wrapper mapping styles

2.3.2 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 [17] 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 2.6.2.1) 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.

2.3.3 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.

2.3.4 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 8.7.3 to enable and disable the asynchronous mapping.

Editors Note 2.1 *JSR-181 currently does not define annotations that can be used to mark a method as being asynchronous.*

2.3.4.1 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.

2.3.4.2 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 2.8 and 2.8.1.

♢ *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 2.8.1.

An application MAY customize this mapping using the `jaxws:method` binding declaration defined in section 8.7.3.

♢ *Conformance (Failed method invocation):* If there is any error prior to invocation of the operation, an implementation MUST throw a `WebServiceException`[1].

2.3.4.3 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 2.3.1.

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.

2.3.4.4 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 2.8 and 2.8.1.

A response bean is mapped from a global element declaration following the rules described in section 2.4. 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

[1] Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation, see section 2.3.4.5.

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`.

2.3.4.5 Faults

Mapping of WSDL faults to service specific exceptions is identical for both asynchronous and synchronous cases, section 2.5 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 (Asychronous 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 6.4).

2.3.4.6 Mapping Examples

Figure 2.2 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.

2.3.4.7 Usage Examples

- Synchronous use.

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

- Asynchronous polling use.

```
1  Service service = ...;
2  StockQuote quoteService = (StockQuote)service.getPort(portName);
3  Response<Float> response = quoteService.getPriceAsync(ticker);
4  while (!response.isDone()) {
5      // do something while we wait
6  }
7  Float quote = response.get();
```

```
<!-- 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 2.2: Asynchronous operation mapping

- Asynchronous callback use.

```
class MyPriceHandler implements AsyncHandler<Float> {
    ...
    public void handleResponse(Response<Float> response) {
        Float price = response.get();
        // do something with the result
    }
}

Service service = ...;
StockQuote quoteService = (StockQuote)service.getPort(portName);
MyPriceHandler myPriceHandler = new MyPriceHandler();
quoteService.getPriceAsync(ticker, myPriceHandler);
```

2.4 Types

Mapping of XML Schema types to Java is described by the JAXB 2.0 specification[10]. The contents of a `wsdl:types` section is passed to JAXB along with any additional type or element declarations (e.g., see section 2.3.4) required to map other WSDL constructs to Java. E.g., section 2.3.4 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 8.5.

♢ *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]. E.g. a parameter that JAXB would map to `JAXBElement<Integer>` is instead be mapped to `Integer`.

JAXB provides support for the SOAP MTOM[24]/XOP[25] 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`).

[2] JAXB maps an element declaration to a Java instance that implements JAXBElement.

2.5 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 2.8 and 2.8.1.

An application MAY customize this mapping using the `jaxws:class` binding declaration defined in section 8.7.4.

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] referred to by that part is mapped to a Java bean, henceforth called a *fault bean*, using the mapping described in section 2.4. 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 6.4.1.

***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 7.2) 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.

2.5.1 Example

Figure 2.3 shows an example of the WSDL fault mapping described above.

[3] WS-I Basic Profile[17] R2205 requires parts to refer to elements rather than types.

```
<!-- 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 2.3: Fault mapping

2.6 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.

2.6.1 General Considerations

R2209 in WS-I Simple SOAP Binding Profile 1.1[26] 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`.

2.6.2 SOAP Binding

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

2.6.2.1 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[17], 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 2.3. 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 2.3. This additional part does not affect eligibility for wrapper style mapping of the message bound by the `soap:body` (see section 2.3.1); 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 implemention 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.

2.6.3 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[27] 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.

2.6.3.1 `mime:content`

Message parts are mapped to method parameters as described in section 2.3 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.

♢ *Conformance (Use of MIME type information):* An implementation MUST support using the `jaxws-:enableMIMEContent` binding declaration defined in section 8.7.5 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] 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 2.3. 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 2.3. These parts are mapped using the non-wrapper style.

Figure 2.4 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[]`.

[4] Multiple `mime:content` elements for the same part indicate a set of permissible alternate types.

```
<!-- 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 2.4: 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[27] when generating the MIME `Content-ID` header field value for a part bound using `mime:content`.

2.7 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 4.1 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 2.8 and 2.8.1.

An application MAY customize the name of the generated service class using the `jaxws:class` binding declaration defined in section 8.7.7.

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 2.0 mandates that two constructors be present on every generated service class.

♢ *Conformance:* A generated service class MUST have a default (i.e. zero-argument) public constructor. This constructor MUST call the protected constructor declared in `javax.xml.ws.Service`, 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:* 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 protected constructor in `javax.xml.ws.Service` passing as arguments the WSDL location and the service name values with which it was invoked.

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

`get`*PortName*`()` 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.

♢ *Conformance (Failed getPort Method):* A generated `get`*PortName* 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 2.8, 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 method for a port using the `jaxws:method` binding declaration defined in section 8.7.8.

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.

2.7.1 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(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name="StockQuoteHTTPPort")
    public StockQuoteProvider getStockQuoteHTTPPort() {
    return (StockQuoteProvider)super.getPort("StockQuoteHTTPPort",
                                             StockQuoteProvider.class);
    }

    @WebEndpoint(name="StockQuoteSMTPPort")
    public StockQuoteProvider getStockQuoteSMTPPort() {
    return (StockQuoteProvider)super.getPort("StockQuoteSMTPPort",
                                             StockQuoteProvider.class);
    }
}
```

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

2.8 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.

2.8.1 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 4.2), the prefix "`_`" is added to the mapped method.

Chapter 3

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 7.

♢ *Conformance (Standard annotations):* An implementation MUST support the use of annotations defined in section 7 to customize the Java to WSDL 1.1 mapping.

3.1 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].

3.1.1 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.

3.2 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 7.11.1) 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.

3.3 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 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 pratice, 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.

3.4 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 7.11.1).
- Any of its methods MAY carry a `javax.jws.WebMethod` annotation (see 7.11.2).
- `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 7.11.1) 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 3.2.

Figure 3.1 shows an example of a Java SEI and the corresponding `wsdl:portType`.

3.4.1 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.

3.5 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 7.11.2) 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 3.5.1 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 3.7 for further details on exception mapping.

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 section 3.7.

Each `wsdl:message` element has one of the following[1]:

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 3.6 describes the mapping from Java methods and their parameters to corresponding global element declarations and named types in the `wsdl:types` section.

3.5.1 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 7.11.3) 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.

3.6 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

[1]The `javax.jws.WebParam` and `javax.jws.WebResult` annotations can introduce additional parts into messages when the `header` element is `true`.

```
1  // Java
2  package com.example;
3  @WebService
4  public interface StockQuoteProvider {
5      float getPrice(String tickerSymbol)
6          throws TickerException;
7  }
8
9  <!-- WSDL extract -->
10 <types>
11     <xsd:schema targetNamespace="...">
12         <!-- element declarations -->
13         <xsd:element name="getPrice"
14             type="tns:getPriceType"/>
15         <xsd:element name="getPriceResponse"
16             type="tns:getPriceResponseType"/>
17         <xsd:element name="TickerException"
18             type="tns:TickerExceptionType"/>
19
20         <!-- type definitions -->
21         ...
22     </xsd:schema>
23 </types>
24
25 <message name="getPrice">
26     <part name="getPrice" element="tns:getPrice"/>
27 </message>
28
29
30 <message name="getPriceResponse">
31     <part name="getPriceResponse" element="tns:getPriceResponse"/>
32 </message>
33
34
35 <message name="TickerException">
36     <part name="TickerException" element="tns:TickerException"/>
37 </message>
38
39
40 <portType name="StockQuoteProvider">
41     <operation name="getPrice">
42         <input message="tns:getPrice"/>
43         <output message="tns:getPriceResponse"/>
44         <fault message="tns:TickerException"/>
45     </operation>
46 </portType>
```

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

```
1  // Java
2  package com.example;
3  @WebService
4  public interface StockQuoteProvider {
5      float getPrice(String tickerSymbol)
6          throws TickerException;
7  }
8
9  <!-- WSDL extract -->
10 <types>
11     <xsd:schema targetNamespace="...">
12         <!-- element declarations -->
13         <xsd:element name="TickerException"
14             type="tns:TickerExceptionType"/>
15
16         <!-- type definitions -->
17         ...
18     </xsd:schema>
19 </types>
20
21 <message name="getPrice">
22     <part name="tickerSymbol" type="xsd:string"/>
23 </message>
24
25
26 <message name="getPriceResponse">
27     <part name="return" type="xsd:float"/>
28 </message>
29
30
31 <message name="TickerException">
32     <part name="TickerException" element="tns:TickerException"/>
33 </message>
34
35
36 <portType name="StockQuoteProvider">
37     <operation name="getPrice">
38         <input message="tns:getPrice"/>
39         <output message="tns:getPriceResponse"/>
40         <fault message="tns:TickerException"/>
41     </operation>
42 </portType>
```

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

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.

3.6.1 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 7.11.4) 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 7.11.4) 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 7.11.4) 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` elment 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.

3.6.2 Use of JAXB

JAXB 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`.

3.6.2.1 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*).

♢ *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 7.3 and 7.4) 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 7.3 and 7.4) 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 distiguish filenames based on case so comparisons MUST ignore case.

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.

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.

```
1  float getPrice(@WebParam(name="tickerSymbol") String sym);
2
3  @XmlRootElement(name="getPrice", targetNamespace="...")
4  @XmlType(name="getPrice", targetNamespace="...")
5  @XmlAccessorType(AccessType.FIELD)
6  public class GetPrice {
7      @XmlElement(name="tickerSymbol", targetNamespace="")
8      public String tickerSymbol;
9  }
10
11 @XmlRootElement(name="getPriceResponse", targetNamespace="...")
12 @XmlType(name="getPriceResponse", targetNamespace="...")
13 @XmlAccessorType(AccessType.FIELD)
14 public class GetPriceResponse {
15     @XmlElement(name="return", targetNamespace="")
16     public float _return;
17 }
```

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 3.6.1) 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 3.6.1) 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 3.1.

3.6.2.2 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 3.1.

3.6.2.3 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`[2].

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.

[2]Use of `RPC` style requires use of `WRAPPED` parameter style. Deviations from this is an error

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`.

3.7 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.

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 2.5 (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 2.5, 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 `getClass` 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 whose `name` property is set to the name of the exception and whose `namespace` property is set to the namespace name mapped from the exception package name. 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 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 distiguish 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(Sting ticker) { ... }
    public UnknownTicker(Sting ticker, String message) { ... }
    public UnknownTicker(Sting 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.

3.8 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 7.8), if present, otherwise the default is the SOAP 1.1/HTTP binding (see 10).

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.

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

3.8.1 Interface

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

```
<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 `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 3.11.

3.8.2 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.

3.9 Generics

In JAX-WS when starting from Java and if generics are used in the document wrapped case, impelementations are required to use type erasure when generating the request /response wrapper beans and exception beans except in the case of `Collections`. 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 snippet shows the result of erasure on a wrapper bean that is generated when using generics:

```
public <T extends Bar> T echoBar(T  value) {
    return value;
}
```

The generated wrapper bean would be

```
@XmlRootElement(name = "echoBar", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "echoBar", namespace = "...")
public class EchoBar {

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

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

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

}
```

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

```
public List<Bar> echoBarList(List<Bar> list) {
    return list;
}
```

The generated wrapper bean would be

```
@XmlRootElement(name = "echoBarList", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "echoBarList", namespace = "...")
public class EchoBarList {

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

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

    public void setArg0(List<Bar> 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 Bar> echoWildcardBar(List<? extends Bar> list) {
    return list;
}
```

The generated wrapper bean would be

```
@XmlRootElement(name = "echoWildcardBar", namespace = "...")
@XmlAccessorType(AccessType.FIELD)
@XmlType(name = "echoWildcardBar", namespace = "...")
public class EchoWildcardBar {

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

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

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

3.10 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

3.10.1 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 3.11).

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`.

3.10.2 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

3.11 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 3.2.

In mapping a `@WebService`-annotated class (see 3.3) 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 [13] 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 "Service"suffix 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.

♢ *Conformance (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 3.8).

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 3.10.1.

Chapter 4

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.

4.1 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 4.1.1. `Service` instances provide facilities to:

- Create an instance of a proxy via one of the `getPort` methods. See section 4.2.3 for information on proxies.
- Create a `Dispatch` instance via the `createDispatch` method. See section 4.3 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 9.2.1.1).
- Configure the `java.util.concurrent.Executor` to be used for asynchronous invocations (see section 4.1.4).

♢ *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 6.3.

4.1.1 Service Usage

4.1.1.1 Dynamic case

In the dynamic case, when nothing is generated, a J2SE service client uses `Service.create` to create `Service` instances, the following code illustrates this process.

```
1  URL wsdlLocation = new URL("http://example.org/my.wsdl");
2  QName serviceName = new QName("http://example.org/sample", "MyService");
3  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.

♢ *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`).

4.1.1.2 Static case

When starting from a WSDL document, a concrete service implementation class MUST be generated as defined in section 2.7. The generated implementation class will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a `java.net.URL`) and the service name (a `javax.xml.namespace.QName`) respectively.

When using the no-argument constructor, the 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:

```
1  // Generated Service Class
2
3  @WebServiceClient(name="StockQuoteService",
4                    targetNamespace="http://example.com/stocks",
5                    wsdlLocation="http://example.com/stocks.wsdl")
6  public class StockQuoteService extends javax.xml.ws.Service {
7      public StockQuoteService() {
8      super(new URL("http://example.com/stocks.wsdl"),
9                new QName("http://example.com/stocks",
10                         "StockQuoteService"));
11     }
12
13     public StockQuoteService(String wsdlLocation, QName serviceName) {
14     super(wsdlLocation, serviceName);
15     }
16
17     ...
18 }
```

4.1.2 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 6.3) 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 6.2.2) 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);
    }

    ...
}
```

4.1.3 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 9 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.

4.1.4 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.

4.2 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.

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 6.1. The following subsection describes the function and use of context with `BindingProvider` instances.

4.2.1 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 9.4.1) prior to invoking any handlers (see chapter 9) 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.

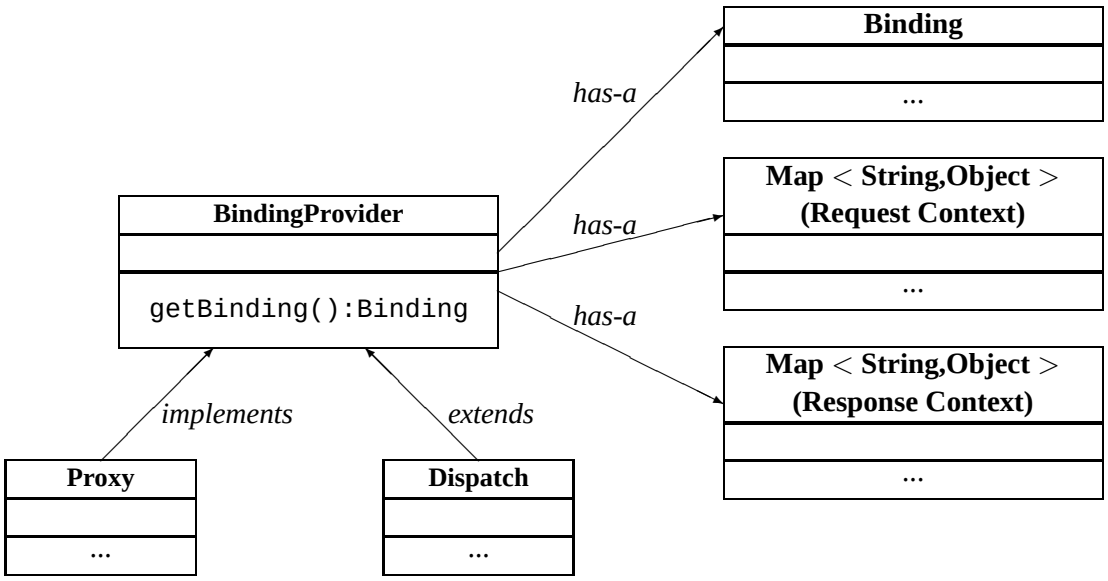

Figure 4.1: Binding Provider Class Relationships

♢ *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 9) 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.*

4.2.1.1 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`			

Continued on next page

Table 4.1 – continued from previous page

Name	Type	Mandatory	Description
`.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 `javax.xml.ws.soap-.http.soapaction.use` 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 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 4.1.

4.2.1.2 Additional Properties

♢ *Conformance (Additional context properties):* Implementations MAY define additional implementation specific properties not listed in 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.

4.2.2 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 2.3.4 and 4.3.3 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 4.1.4 for more information on configuring the `Executor` to be used.

4.2.3 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.

♢ *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.

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 2 (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`).

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).

4.2.3.1 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.

```
1  javax.xml.ws.Service service = ...;
2  com.example.StockQuoteProvider proxy = service.getPort(portName,
3      com.example.StockQuoteProvider.class)
4  javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider)proxy;
5  Map<String,Object> context = bp.getRequestContext();
6  context.setProperty("javax.xml.ws.session.maintain", Boolean.TRUE);
7  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.

4.2.4 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 implemention 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 6.4.1 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 `WebService-Exception` 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.

4.3 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 11) in payload mode, the HTTP request and response entity bodies must contain XML directly, MIME wrapper with an XML root part or the `Source` passed to the invoke method may be `null`. 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 11) 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 mssage 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 10) 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 11) in message mode.

4.3.1 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 4.2.1.1 for a list of standard properties.

4.3.2 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.

♢ *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[1] during the remote operation invocation.

See section 10.4.1 for additional SOAP/HTTP requirements.

4.3.3 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.

[1] 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.

♢ *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.

4.3.4 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`.

4.3.5 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.

4.3.5.1 Synchronous, Payload-Oriented

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

4.3.5.2 Synchronous, Message-Oriented

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

4.3.5.3 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’.

4.3.5.4 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();
```

4.3.5.5 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);
```

4.4 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 [28]. 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 5.2.5) are subject to the requirements in this section, resulting in catalog-based URI resolutions.

Chapter 5

Service APIs

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

5.1 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 3.4 describes the requirements that a Java interface must meet to qualify as a JAX-WS SEI. Section 2.2 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.

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 7.7).

`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.

5.1.1 Invocation

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

5.1.1.1 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 6.4.1. 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.

5.1.2 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 5.3) 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`.

5.1.3 Examples

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

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

```
1  @WebServiceProvider
2  @ServiceMode(value=Service.Mode.MESSAGE)
3  public class MyService implements Provider<SOAPMessage> {
4      public MyService() {
5      }
6
7      public SOAPMessage invoke(SOAPMessage request) {
8          return request;
9      }
10 }
```

Simple static reply, reply message contains a fixed acknowlegment element

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

```
    }

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

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);
    }
}
```

5.2 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.

5.2.1 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(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.

`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 if 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.

These methods MUST delegate the creation of Endpoint to the `javax.xml.ws.spi.Provider` SPI class (see 6.2) 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 3 or an instance of a class annotated with the `WebServiceProvider` annotation and implementing the `Provider` interface (see 5.1).

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:

```
1  // assume Test is an endpoint implementation class annotated with @WebService
2  Test test = new Test();
3  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):* 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 10) 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 5.2.3.

Endpoint implementors MAY use the `WebServiceContext` facility (see 5.3) 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 [29].

5.2.2 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 10) or HTTP (see 11) 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 5.2.4).

♢ *Conformance (WSDL Publishing):* An `Endpoint` that uses the SOAP 1.1/HTTP binding (see 10) 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 5.2.3.

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 5.2.7.

5.2.2.1 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.

```
1  // assume Test is an endpoint implementation class annotated with @WebService
2  Test test = new Test();
3  HttpServer server = HttpServer.create(new InetSocketAddress(8080),10);
4  server.setExecutor(Executor.newFixedThreadPool(10));
5  server.start();
6  HttpContext context = server.createContext("/test");
7  Endpoint endpoint = Endpoint.create(SOAPBinding.SOAP11HTTP_BINDING, test);
8  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.

5.2.3 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`.

5.2.4 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 7).

♢ *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`).

5.2.5 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 5.2.5.3.

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 5.2.4. 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 [14] 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.

5.2.5.1 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 5.2.5.3 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.

Note: *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 3 and those in the JAXB specification [10].

5.2.5.2 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.

5.2.5.3 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 7) 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 5.2.5.4 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 4.4) 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 4.4, 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:

5.2.5.4 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. 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**.

5.2.5.5 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**.

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.

5.2.5.6 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.*

5.2.6 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 7.7), 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.

5.2.7 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.

5.3 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`:

```
1  @WebService
2  public class Test {
3      @Resource
4      private WebServiceContext context;
5
6      public String reverse(String inputString) { ... }
7  }
```

The `javax.annotation.Resource` annotation defined by JSR-250 [29] 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 [30].

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.*

5.3.1 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 9). 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.

Chapter 6

Core APIs

This chapter describes the standard core APIs that may be used by both client and server side applications.

6.1 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 10 describes the JAX-WS SOAP binding; chapter 11 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 4.2, 5.2).

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 10 and 11). 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 9.2.1).

♢ *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`

6.2 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 4.1) and `Endpoint` (see 5.2) 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.

6.2.1 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, 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. If a resource with the name of `META-INF/services/javax.xml.ws.spi.Provider` exists, then its first line, if present, is used as the UTF-8 encoded name of the implementation class.
2. If the `${java.home}/lib/jaxws.properties` file exists and it is readable by the `java.util.Properties.load(InputStream)` method and it contains an entry whose key is `javax.xml.ws.spi.Provider`, then the value of that entry is used as the name of the implementation class.
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 default implementation class name is used.

6.2.2 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.

`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 if 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.

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 3, 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 5.1.

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 5.2.1.

♢ *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`.

6.2.3 Creating ServiceDelegate Objects

`javax.xml.ws.spi.ServiceDelegate` 6.3 can be created using the following method on `Provider`:

`createServiceDelegate(URL wsdlDocumentLocation, QName serviceName, Class 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 2.7. 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.

6.3 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` 4.1 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`.

6.4 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.*

6.4.1 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 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.

6.4.1.1 Client Side Example

```
try {
    response = dispatch.invoke(request);
}
catch (SOAPFaultException e) {
    QName soapFaultCode = e.getFault().getFaultCodeAsQName();
    ...
}
```

6.4.1.2 Server Side Example

```
public void endpointOperation() {
    ...
    if (someProblem) {
        SOAPFault fault = soapBinding.getSOAPFactory().createFault(
            faultcode, faultstring, faultactor, detail);
        throw new SOAPFaultException(fault);
    }
    ...
}
```

6.4.2 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.

Chapter 7

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 [13]).

♢ *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.

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.

7.1 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
`value`	The service mode, one of javax.xml.ws.Service.Mode. MESSAGE or javax.xml.ws.Service.Mode.PAYLOAD. MESSAGE means that the whole protocol message will be handed to the provider instance, PAYLOAD that only the payload of the protocol message will be handed to the provider instance.	javax.xml.ws-.Service.Mode-.PAYLOAD

The `ServiceMode` annotation type is marked `@Inherited`, so the annotation will be inherited from the superclass.

7.2 javax.xml.ws.WebFault

The `WebFault` annotation is used when mapping WSDL faults to Java exceptions, see section 2.5. 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
`name`	The local name of the element	""
`targetNamespace`	The namespace name of the element	""
`faultBean`	The fully qualified name of the fault bean class	""

7.3 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
`localName`	The local name of the element	""
`targetNamespace`	The namespace name of the element	""
`className`	The name of the wrapper class	""

7.4 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
`localName`	The local name of the element	""
`targetNamespace`	The namespace name of the element	""
`className`	The name of the wrapper class	""

7.5 javax.xml.ws.WebServiceClient

The `WebServiceClient` annotation is specified on a generated service class (see 2.7). 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: `WebServiceClient` properties.

Property	Description	Default
`name`	The local name of the service	""
`targetNamespace`	The namespace name of the service	""
`wsdlLocation`	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 4.4.

7.6 javax.xml.ws.WebEndpoint

The `WebEndpoint` annotation is specified on the `get`*PortName*`()` methods of a generated service class (see 2.7). 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
`name`	The local name of the port	""

7.6.1 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 Interface
@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) {

    }
    @WebEndpoint(name="StockQuoteHTTPPort")
    public StockQuoteProvider getStockQuoteHTTPPort() {
    return (StockQuoteProvider)super.gePort(portName, StockQuoteProvider.class);
    }

    @WebEndpoint(name="StockQuoteSMTPPort")
    public StockQuoteProvider getStockQuoteSMTPPort() {
    return (StockQuoteProvider)super.getPort(portName, StockQuoteProvider.class);
    }
}
```

7.7 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 5.1) 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
`wsdlLocation`	The URL for the WSDL description	""
`serviceName`	The name of the service	""
`portName`	The name of the port	""
`targetNamespace`	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 4.4.

7.8 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
`value`	The binding ID (a URI)	""

The default binding for an endpoint is the SOAP 1.1/HTTP one (see chapter 10).

7.9 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 [29].

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 [30].

Table 7.9: `WebServiceRef` properties.

Property	Description	Default
`name`	The name identifying the Web service reference.	""
`wsdlLocation`	A URL pointing to the location of the WSDL document for the service being referred to.	""
`type`	The resource type as a Java class object	`Object.class`
`value`	The service type as a Java class object	`Object.class`
`mappedName`	A product specific name that this resource should be mapped to.	""

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. (It's a name in 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.

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 4.4.

7.9.1 Example

The following shows both uses of the `WebServiceRef` annotation.

```

// Generated Service Interface

@WebServiceClient(name="StockQuoteService",
                  targetNamespace="...",
                  wsdlLocation="...")
public interface StockQuoteService extends javax.xml.ws.Service {
    @WebEndpoint(name="StockQuoteHTTPPort")
    StockQuoteProvider getStockQuoteHTTPPort();

    @WebEndpoint(name="StockQuoteSMTPPort")
    StockQuoteProvider getStockQuoteSMTPPort();
}

// Generated SEI

@WebService(name="StockQuoteProvider",
            targetNamespace="...")
public interface StockQuoteProvider {
    Double getStockQuote(String ticker);
}

// Sample client code

@Stateless
public ClientComponent {

    // WebServiceRef using the generated service interface type
    @WebServiceRef
    public StockQuoteService stockQuoteService;

    // WebServiceRef using the SEI type
    @WebServiceRef(StockQuoteService.class)
```

```
34      private StockQuoteProvider stockQuoteProvider;
35
36     // other methods go here...
37  }
```

7.10 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 limition 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 [29].

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 [30].

Table 7.10: `WebServiceRefs` properties.

Property	Description	Default
`value`	An array of `WebServiceRef` annotations, each defining a web service reference.	{}

7.10.1 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.

```
1
2   @WebServiceRefs({@WebServiceRef(name="accounting"
3                                   type=AccountingPortType.class,
4                                   value=AccountingService.class),
5                    @WebServiceRef(name="payroll",
6                                   type=PayrollService.class)})
7   @Stateless
8   public MyComponent {
9
10     // methods using the declared resources go here...
11  }
```

7.11 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 [13].

As a convenience to the reader, the following sections reproduce the definition of the JSR-181 annotations applicable to JAX-WS.

7.11.1 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 4.4.

7.11.2 javax.jws.WebMethod

```
@Target({METHOD})
public @interface WebMethod {
  String operationName() default "";
  String action() default "" ;
  boolean exclude() default false;
};
```

7.11.3 javax.jws.OneWay

```
@Target({METHOD})
public @interface Oneway {
};
```

7.11.4 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 "";
};
```

7.11.5 javax.jws.WebResult

```
@Target({METHOD})
```

```
public @interface WebResult {
  String name() default "return";
  String targetNamespace() default "";
  boolean header() default false;
  String partName() default "";
};
```

7.11.6 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;
}
```

7.11.7 javax.jws.HandlerChain

```
@Target({TYPE})
public @interface HandlerChain {
    String file();
    String name() default "";
}
```

Chapter 8

Customizations

This chapter describes a standard customization facility that can be used to customize the WSDL 1.1 to Java binding defined in section 2.

8.1 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.

♢ *Conformance (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.

♢ *Conformance (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`.

8.2 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 8.4). The second approach consists in embeddeding binding declarations directly inside a WSDL document (see 8.3).

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`.

8.3 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[17].

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` amongs its children.

8.3.1 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`.

8.4 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="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"/>
      <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>
```

Figure 8.2: Sample WSDL document with embedded binding declarations

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 8.3). 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 8.5 for more information on processing JAXB binding declarations.

8.4.1 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 8.3.1.

```
<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

8.5 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.

```
1   <wsdl:definitions targetNamespace="..." xmlns:tns="..." xmlns:stns="...">
2     <wsdl:types>
3       <xs:schema targetNamespace="http://example.org/bar">
4         <xs:element name="setLastTradePrice">
5           <xs:complexType>
6             <xs:sequence>
7               <xs:element name="tickerSymbol" type="xs:string"/>
8               <xs:element name="lastTradePrice" type="xs:float"/>
9             </xs:sequence>
10          </xs:complexType>
11        </xs:element>
12        <xs:element name="setLastTradePriceResponse">
13          <xs:complexType>
14            <xs:sequence/>
15          </xs:complexType>
16        </xs:element>
17      </xs:schema>
18    </wsdl:types>
19
20    <wsdl:message name="setLastTradePrice">
21      <wsdl:part name="setPrice" element="stns:setLastTradePrice"/>
22    </wsdl:message>
23
24    <wsdl:message name="setLastTradePriceResponse">
25      <wsdl:part name="setPriceResponse"
26        type="stns:setLastTradePriceResponse"/>
27    </wsdl:message>
28
29    <wsdl:portType name="StockQuoteUpdater">
30      <wsdl:operation name="setLastTradePrice">
31        <wsdl:input message="tns:setLastTradePrice"/>
32        <wsdl:output message="tns:setLastTradePriceResponse"/>
33      </wsdl:operation>
34    </wsdl:portType>
35  </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.*

8.6 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.

8.7 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.

8.7.1 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>
```

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. disbled) 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.

8.7.2 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>?
```

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`.

8.7.3 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"/>*
```

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.

8.7.4 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>
```

Semantics

class/@name The name of the generated exception class for this fault.

class/javadoc/text() 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.

8.7.5 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>
```

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.

8.7.6 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">?
```

```
11          <jaxws:javadoc>xs:string</jaxws:javadoc>?
12        </jaxws:class>
13      </jaxws:exception>
```

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.

8.7.7 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.

```
1       <jaxws:class name="xs:string">?
2         <jaxws:javadoc>xs:string</jaxws:javadoc>?
3       </jaxws:class>
```

Semantics

class/@name The name of the generated service interface.

class/javadoc/text() Class-level javadoc string.

8.7.8 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.

```
1       <jaxws:method name="xs:string">?
2         <jaxws:javadoc>xs:string</jaxws:javadoc>?
3       </jaxws:method>
4
5       <jaxws:provider/>?
```

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 9

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.

9.1 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 9.1). Protocol bindings can extend the handler framework to provide protocol specific functionality; chapter 10 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.

9.1.1 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

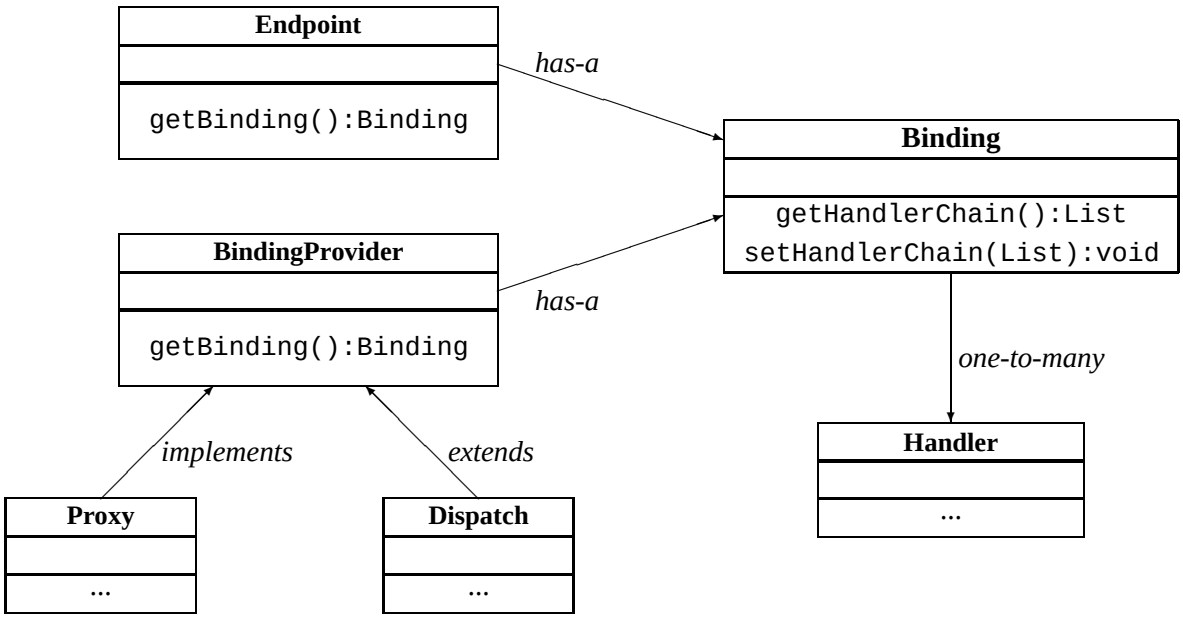

Figure 9.1: Handler architecture

message. Protocol handlers are handlers that implement any interface derived from `javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

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`.

9.1.2 Binding Responsibilities

The following subsections describe the responsibilities of the protocol binding when hosting a handler chain.

9.1.2.1 Handler and Message Context Management

The binding is responsible for instantiation, invocation, and destruction of handlers according to the rules specified in section 9.3. The binding is responsible for instantiation and management of message contexts according to the rules specified in section 9.4

♢ *Conformance (Logical handler support):* All binding implementations MUST support logical handlers (see section 9.1.1) being deployed in their handler chains.

♢ *Conformance (Other handler support):* Binding implementations MAY support other handler types (see section 9.1.1) being deployed in their handler chains.

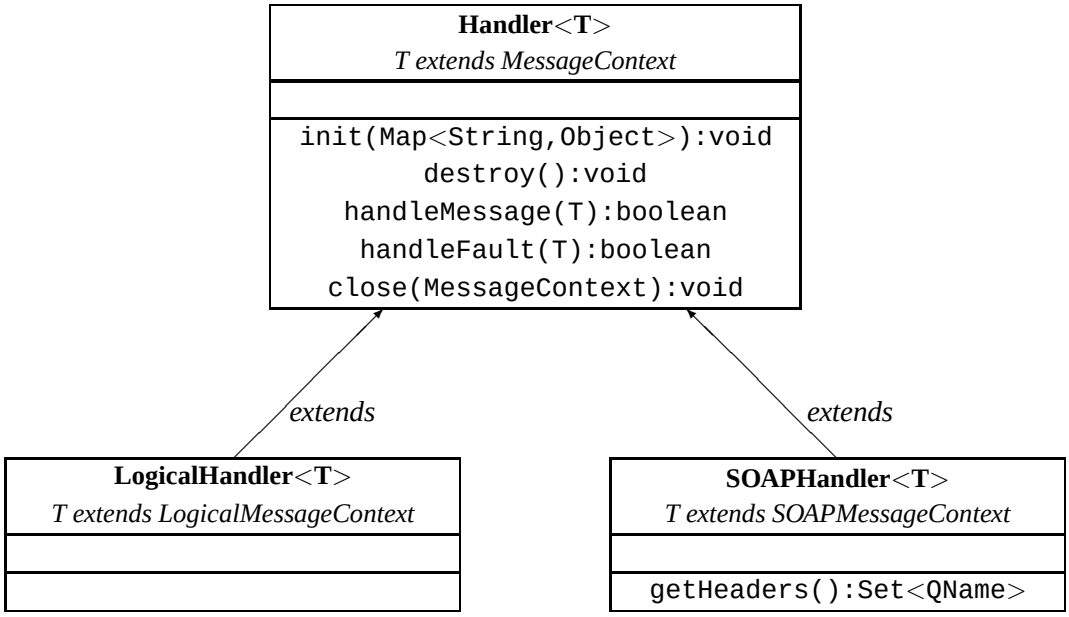

Figure 9.2: Handler class hierarchy

♢ *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.

9.1.2.2 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.

9.1.2.3 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 9.3.2.

Outbound exceptions[1] 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

[1]Outbound exceptions are exceptions thrown by a handler that result in the message direction being set to outbound according to the rules in section 9.3.2.

particular protocol, section 10.2.2 describes the mechanism for the SOAP 1.1 binding. Inbound exceptions are passed to the binding provider.

9.2 Configuration

Handler chains may be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

9.2.1 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.

9.2.1.1 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.

9.2.1.2 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 9.3.2 describes how the handler order relates to the order of handler execution for inbound and outbound messages.

9.2.1.3 javax.jws.HandlerChain annotation

The `javax.jws.HandlerChain` annotation defined by JSR-181 [13] may be used to specify in a declarative way the handler chain to use for a service.

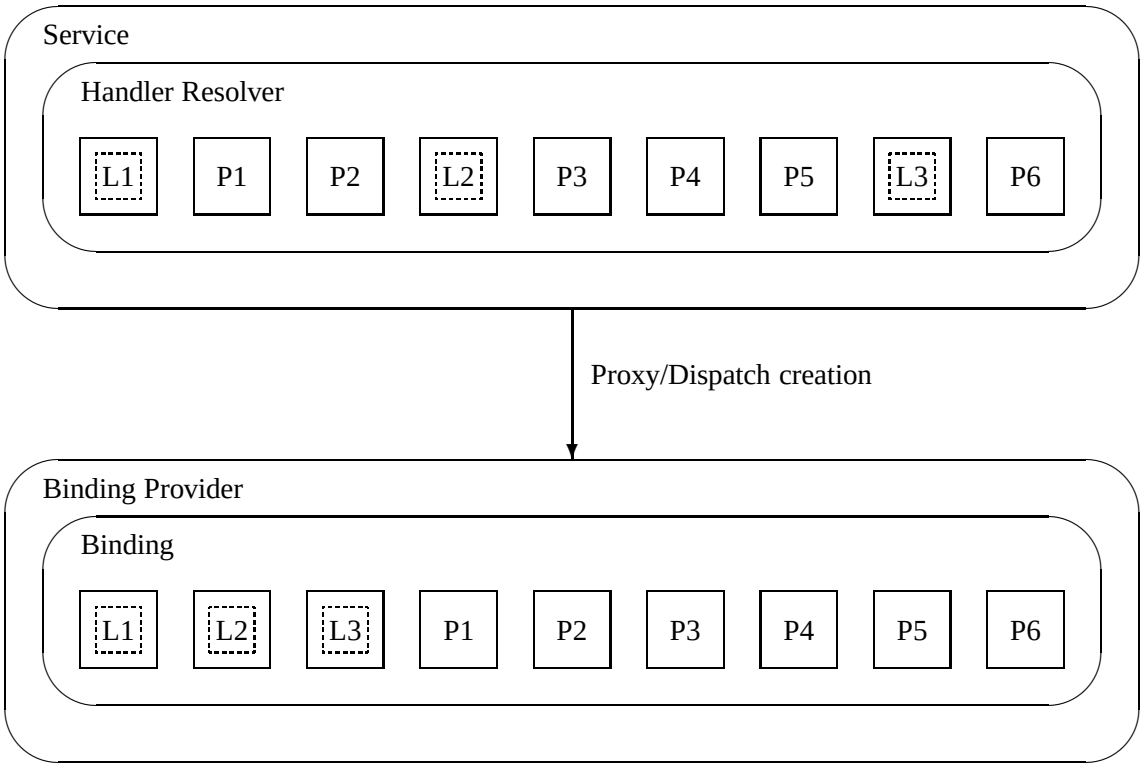

Figure 9.3: Handler ordering, L*n* and P*n* represent logical and protocol handlers respectively.

When used in conunction 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 9.2.1.1).

♢ *Conformance (Handler resolver for a HandlerChain annotation):* For a generated service class (see 2.7) 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.

9.2.1.4 javax.xml.ws.Binding

The `Binding` interface is an abstraction of a JAX-WS protocol binding (see section 6.1 for more details). As described above, the handler chain initially configured on an instance is a snapshot of the applicable handlers

```
1  @WebService
2  @HandlerChain(file="sample_chain.xml")
3  public class MyService {
4      ...
5  }
```

Figure 9.4: Use of the `HandlerChain` annotation

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.

9.2.2 Deployment Model

JAX-WS defines no standard deployment model for handlers. Such a model is provided by JSR 109[14] "Implementing Enterprise Web Services".

9.3 Processing Model

This section describes the processing model for handlers within the handler framework.

9.3.1 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 [29]

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 [29] 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.

9.3.2 Handler Execution

As described in section 9.2.1.2, 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 9.4) 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 $H_1 \ldots H_6$ 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.

9.3.2.1 `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 9.1.2.2) 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*[2] or not:

Response The message direction is reversed, the runtime invokes `handleMessage` on the next[3] handler or dispatches the message (see section 9.1.2.2) 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 9.1.2.2).

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[4], and the runtime invokes `handleFault` on the next[4] handler or dispatches the message (see section 9.1.2.2) 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 9.1.2.3).

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 9.1.2.3).

No response Normal message processing stops, `close` is called on each previously invoked handler in the chain, the exception is dispatched (see section 9.1.2.3).

[2]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.

[3]Next in this context means the next handler taking into account the message direction reversal

[4]The handler may have already converted the message to a fault message, in which case no change is made.

9.3.2.2 `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 9.1.2.2) 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 9.1.2.2).

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 9.1.2.3).

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 9.1.2.3).

9.3.2.3 `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 9.3.2).

9.3.3 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 9.4.

9.4 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 9.4.1 and 9.4.2 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 10.3 describes the message context subtype for the JAX-WS SOAP binding.

9.4.1 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 4.2.1) and service endpoint implementations. The defaultscope 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.

9.4.1.1 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.

9.4.2 javax.xml.ws.handler.LogicalMessageContext

Logical handlers (see section 9.1.1) 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 10.1.1.2, 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 11 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 imple-

Table 9.2: Standard `MessageContext` properties.

Name	Type	Mandatory	Description
`javax.xml.ws.handler.message`			
`.outbound`	Boolean	Y	Specifies the message direction: `true` for outbound messages, `false` for inbound messages.
`javax.xml.ws.binding.attachments`			
`.inbound`	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 `DataHandler` for the attachment data. Bindings describe how to carry attachments with messages.
`.outbound`	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 `DataHandler` for the attachment data. Bindings describe how to carry attachments with messages.
`javax.xml.ws.wsdl`			
`.description`	URI	N	A resolvable URI that may be used to obtain access to the WSDL for the endpoint.
`.service`	`QName`	N	The name of the service being invoked in the WSDL.
`.port`	QName	N	The name of the port over which the current message was received in the WSDL.
`.interface`	QName	N	The name of the port type to which the current message belongs.
`.operation`	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`			
`.headers`	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.
`.method`	`String`	Y	The HTTP method for the request message.
`.querystring`	`String`	Y	The HTTP query string for the request message, or `null` if the request does not have any. If the address specified using the javax.xml.ws.service.endpoint.address in the BindingProvider contains a query string and if the querystring property is set by the client it will override the existing query string in the javax.xml.ws.service.endpoint.address property. The value of the property does not include the leading "?" of the query string in it. This property is only used with HTTP binding.
`.pathinfo`	`String`	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 "/" character.
`javax.xml.ws.http.response`			
`.headers`	Map<String,List<String>>	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`	Integer	Y	The HTTP response status code.

Table 9.4: Standard Servlet Container-Specific `MessageContext` properties.

Name	Type	Mandatory	Description
`javax.xml.ws.servlet`			
`.context`	javax.servlet.ServletContext	Y	The `ServletContext` object belonging to the web application that contains the endpoint.
`.request`	javax.servlet.http.HttpServletRequest	Y	The `HttpServletRequest` object associated with the request currently being served.
`.response`	javax.servlet.http.HttpServletResponse	Y	The `HttpServletResponse` object associated with the request currently being served.

mentation 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 11).

9.4.3 Relationship to Application Contexts

Client side binding providers have methods to access contexts for outbound and inbound messages. As described in section 4.2.1 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 5, 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 10

SOAP Binding

This chapter describes the JAX-WS SOAP binding and its extensions to the handler framework (described in chapter 9) for SOAP message processing.

10.1 Configuration

A SOAP binding instance requires SOAP specific configuration in addition to that described in section 9.2. The additional information can be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

10.1.1 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 5.2), but are otherwise expected to be configured using annotations or deployment metadata.

10.1.1.1 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`.

10.1.1.2 SOAP Handlers

The handler chain for a SOAP binding is configured as described in section 9.2.1. 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` 9.2 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 properites in consideration above in the `MessageContext` after invoking the first `SOAPHandler` are ignored. The `SOAPHandler` however may change the properties in the `MessageContext`

♢ *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.

10.1.1.3 SOAP Headers

The SOAP headers understood by a handler are obtained using the `getHeaders` method of `SOAPHandler`.

10.1.2 Deployment Model

JAX-WS defines no standard deployment model for handlers. Such a model is provided by JSR 109[14] "Implementing Enterprise Web Services".

10.2 Processing Model

The SOAP binding implements the general handler framework processing model described in section 9.3 but extends it to include SOAP specific processing as described in the following subsections.

10.2.1 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 9.3.2). 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 `Handlers` 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 `QNames` that satisfy at least one of the following conditions:
   (a) that are mapped to method parameters in the service endpoint interface;
   (b) are members of `SOAPHandler.getHeaders()` for each `SOAPHandler` in the set obtained in step 2;
   (c) are directly supported by the binding instance.
4. 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 1.
5. For each header in the set obtained in step 4, the header is understood if its QName is in the set identified in step 3.
6. If every header in the set obtained in step 4 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 10.2.2).

   No response The binding dispatches the SOAP must understand exception (see section 10.2.2).

10.2.2 Exception Handling

The following subsections describe SOAP specific requirements for handling exceptions thrown by handlers and service endpoint implementations.

10.2.2.1 Handler Exceptions

A binding is responsible for catching runtime exceptions thrown by handlers and following the processing model described in section 9.3.2. 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 10.2.2.3. 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 9.3.2.

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.

10.2.2.2 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 6.4.1.

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 9.3.2.

Section 10.2.2.3 describes the rules for mapping an exception to a SOAP fault.

10.2.2.3 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()`[1]
  2. `env:Server` (`Subcode` omitted for SOAP 1.2).
- `faultstring` (`Reason/Text`
  1. `SOAPFaultException.getFault().getFaultString()`[1]
  2. `Exception.getMessage()`
  3. `Exception.toString()`
- `faultactor` (`Role` in SOAP 1.2)
  1. `SOAPFaultException.getFault().getFaultActor()`[1]
  2. Empty
- `detail` (`Detail` in SOAP 1.2)
  1. Serialized service specific exception (see *WrapperException*.`getFaultInfo()` in section 2.5)
  2. `SOAPFaultException.getFault().getDetail()`[1]

10.3 SOAP Message Context

SOAP handlers are passed a `SOAPMessageContext` when invoked. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.

10.4 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.

10.4.1 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[31] as clarified by the WS-I Basic Profile[17], WS-I Simple SOAP Binding Profile[26] and WS-I Attachment Profile[27].

[1] If the exception is a `SOAPFaultException` or has a cause that is a `SOAPFaultException`.

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].

10.4.1.1 MTOM

♢ *Conformance (SOAP MTOM Support):* An implementation MUST support MTOM[24][1].

`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`.

[1] JAX-WS inherits the JAXB support for the SOAP MTOM[24]/XOP[25] mechanism for optimizing transmission of binary data types, see section 2.4.

10.4.1.2 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.

10.4.1.3 Security

Section 4.2.1.1 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.

10.4.1.4 Session Management

Section 4.2.1.1 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 subsequest 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[17] allows a service to use HTTP cookies. However, R1121 recommends that a service should not rely on use of cookies for state management.

♢ *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.

Chapter 11

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.

11.1 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 9.2. 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 only defines APIs for programmatic configuration of client side XML/HTTP bindings – server side bindings are expected to be configured using deployment metadata.

11.1.1.1 HTTP Handlers

The handler chain for an XML/HTTP binding is configured as described in section 9.2.1. 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.

♢ *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.

11.1.2 Deployment Model

JAX-WS defines no standard deployment model for handlers. Such a model is provided by JSR 109[14] "Implementing Enterprise Web Services".

11.2 Processing Model

The XML/HTTP binding implements the general handler framework processing model described in section 9.3.

11.2.1 Exception Handling

The following subsections describe HTTP specific requirements for handling exceptions thrown by handlers and service endpoint implementations.

11.2.1.1 Handler Exceptions

A binding is responsible for catching runtime exceptions thrown by handlers and following the processing model described in section 9.3.2. 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 9.3.2.

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.

11.2.1.2 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 6.4.1.

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 9.3.2.

Section 11.2.1.3 describes the rules for mapping an exception to a HTTP status code.

11.2.1.3 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()`[1]
2. `500`.

11.3 HTTP Support

11.3.1 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.

11.3.2 Security

Section 4.2.1.1 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.

[1] If the exception is a `HTTPException` or has a cause that is a `HTTPException`.

♢ *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.

11.3.3 Session Management

Section 4.2.1.1 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 cokkie and returns it in subsequest 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

2.1 WSDL 1.1 support 9
2.2 Customization required 9
2.3 Annotations on generated classes 9
2.4 Definitions mapping 9
2.5 WSDL and XML Schema import directives 10
2.6 Optional WSDL extensions 10
2.7 SEI naming 10
2.8 `javax.jws.WebService` required 10
2.9 Method naming 11
2.10 `javax.jws.WebMethod` required 11
2.11 Transmission primitive support 11
2.12 Using `javax.jws.OneWay` 11
2.13 Using `javax.jws.SOAPBinding` 11
2.14 Using `javax.jws.WebParam` 11
2.15 Using `javax.jws.WebResult` 11
2.16 Non-wrapped parameter naming 12
2.17 Default mapping mode 12
2.18 Disabling wrapper style 13
2.19 Wrapped parameter naming 13
2.20 Parameter name clash 13
2.21 Using `javax.xml.ws.RequestWrapper` 13
2.22 Using `javax.xml.ws.ResponseWrapper` 13
2.23 Use of `Holder` 16
2.24 Asynchronous mapping required 16

2.25 Asynchronous mapping option 16
2.26 Asynchronous method naming 17
2.27 Asynchronous parameter naming 17
2.28 Failed method invocation 17
2.29 Response bean naming 17
2.30 Asynchronous fault reporting 18
2.31 Asychronous fault cause 18
2.32 JAXB class mapping 20
2.33 JAXB customization use 20
2.34 JAXB customization clash 20
2.35 `javax.xml.ws.WebFault` required 21
2.36 Exception naming 21
2.37 Fault equivalence 21
2.38 Fault equivalence 21
2.39 Required WSDL extensions 23
2.40 Unbound message parts 23
2.41 Duplicate headers in binding 23
2.42 Duplicate headers in message 24
2.43 Use of MIME type information 24
2.44 MIME type mismatch 26
2.45 MIME part identification 26
2.46 `Service` superclass required 26
2.47 Service class naming 26
2.48 `javax.xml.ws.WebServiceClient` required 26
2.49 26
2.50 26
2.51 Failed getPort Method 27
2.52 `javax.xml.ws.WebEndpoint` required 27
3.1 WSDL 1.1 support 29
3.2 Standard annotations 29
3.3 Java identifier mapping 29
3.4 Method name disambiguation 29
3.5 Package name mapping 30
3.6 WSDL and XML Schema import directives 30

3.7 Class mapping 30
3.8 portType naming 31
3.9 Inheritance flattening 31
3.10 Inherited interface mapping 31
3.11 Operation naming 31
3.12 One-way mapping 32
3.13 One-way mapping errors 32
3.14 Parameter classification 35
3.15 Parameter naming 35
3.16 Result naming 35
3.17 Header mapping of parameters and results 35
3.18 Default wrapper bean names 36
3.19 Default wrapper bean package 36
3.20 Wrapper element names 36
3.21 Wrapper bean name clash 36
3.22 Null Values in rpc/literal 39
3.23 Exception naming 39
3.24 Fault bean name clash 40
3.25 Binding selection 40
3.26 SOAP binding support 44
3.27 SOAP binding style required 44
3.28 Service creation 47
3.29 Port selection 47
3.30 Port binding 47
4.1 Service completeness 49
4.2 Service Creation Failure 50
4.3 Use of Executor 52
4.4 Default Executor 52
4.5 Message context decoupling 53
4.6 Required `BindingProvider` properties 54
4.7 Optional `BindingProvider` properties 54
4.8 Additional context properties 54
4.9 Asynchronous response context 55
4.10 Proxy support 55

4.11 Implementing `BindingProvider` 55
4.12 `Service.getPort` failure 55
4.13 Remote Exceptions 56
4.14 Exceptions During Handler Processing 56
4.15 Other Exceptions 56
4.16 `Dispatch` support 56
4.17 Failed `Dispatch.invoke` 58
4.18 Failed `Dispatch.invokeAsync` 58
4.19 Failed `Dispatch.invokeOneWay` 58
4.20 Reporting asynchronous errors 59
4.21 Marshalling failure 59
4.22 Use of the Catalog 61
5.1 Provider support required 63
5.2 Provider default constructor 63
5.3 Provider implementation 63
5.4 WebServiceProvider annotation 63
5.5 Endpoint publish(String address, Object implementor) Method 66
5.6 Default Endpoint Binding 66
5.7 Other Bindings 66
5.8 Publishing over HTTP 67
5.9 WSDL Publishing 67
5.10 Checking `publishEndpoint` Permission 68
5.11 Required Metadata Types 68
5.12 Unknown Metadata 68
5.13 Use of Executor 72
5.14 Default Executor 73
6.1 Read-only handler chains 75
6.2 Concrete `javax.xml.ws.spi.Provider` required 75
6.3 Provider createAndPublishEndpoint Method 76
6.4 Concrete `javax.xml.ws.spi.ServiceDelegate` required 77
6.5 Protocol specific fault generation 77
6.6 Protocol specific fault consumption 78
6.7 One-way operations 78
7.1 Correctness of annotations 79

7.2 Handling incorrect annotations . . . 79
7.3 WebServiceProvider and WebService . . . 82
7.4 JSR-181 conformance . . . 85
8.1 Standard binding declarations . . . 89
8.2 Binding language extensibility . . . 89
8.3 Multiple binding files . . . 92
9.1 Handler framework support . . . 101
9.2 Logical handler support . . . 102
9.3 Other handler support . . . 102
9.4 Incompatible handlers . . . 103
9.5 Incompatible handlers . . . 103
9.6 Handler chain snapshot . . . 104
9.7 HandlerChain annotation . . . 105
9.8 Handler resolver for a HandlerChain annotation . . . 105
9.9 Binding handler manipulation . . . 106
9.10 Handler initialization . . . 107
9.11 Handler destruction . . . 107
9.12 Invoking `close` . . . 109
9.13 Order of `close` invocations . . . 109
9.14 Message context property scope . . . 110
10.1 SOAP required roles . . . 115
10.2 SOAP required roles . . . 116
10.3 Default role visibility . . . 116
10.4 Default role persistence . . . 116
10.5 None role error . . . 116
10.6 Incompatible handlers . . . 116
10.7 Incompatible handlers . . . 116
10.8 Logical handler access . . . 116
10.9 SOAP 1.1 HTTP Binding Support . . . 119
10.10 SOAP 1.2 HTTP Binding Support . . . 120
10.11 SOAP MTOM Support . . . 120
10.12 Semantics of MTOM enabled . . . 120
10.13 MTOM support . . . 120
10.14 SOAP bindings with MTOM disabled . . . 120

10.15 SOAP bindings with MTOM enabled 120
10.16 MTOM on Other SOAP Bindings 120
10.17 One-way operations 121
10.18 HTTP basic authentication support 121
10.19 Authentication properties 121
10.20 URL rewriting support 121
10.21 Cookie support 121
10.22 SSL session support 121
11.1 XML/HTTP Binding Support 123
11.2 Incompatible handlers 123
11.3 Incompatible handlers 124
11.4 Logical handler access 124
11.5 One-way operations 125
11.6 HTTP basic authentication support 125
11.7 Authentication properties 126
11.8 URL rewriting support 126
11.9 Cookie support 126
11.10 SSL session support 126

Appendix B

Change Log

B.1 Changes since Proposed Final Draft

- 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 nillability 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.

B.2 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).

B.3 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 5.2 on the new Endpoint API.
- Added the following new annotation types: @RequestWrapper, @ResponseWrapper, @WebServiceClient, @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 10.2.1 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 4.2.1.1, 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 11.

B.4 Changes Since Early Draft 2

- Renamed ”element” attribute of the jaxws:parameter annotation to ”childParameterName” for clarity, see sections 8.7.3 and 8.7.6.

- Added javax.xml.ws.ServiceMode annotation type, see section 7.1.

- Fixed example of external binding file to use a schema annotation, see section 8.4.

- Modified Dispatch so it can be used with multiple message types and either message payloads or entire messages, see section 4.3.

- Modified Provider so it can be used with multiple message types and either message payloads or entire messages, see section 5.1.

- Added new annotation for generated exceptions, see section 7.2.

- Added default Java package name to WSDL targetNamespace mapping algorithm, see section 3.2.

- Added ordering to properties in request and response beans for doc/lit/wrapped, see section 3.6.2.1.

- Clarified that SEI method should throw JAX-RPC exception with a cause of any runtime exception thrown during local processing, see section 4.2.4.

- Removed requirement that SEIs MUST NOT have constants, see section 3.4.

- Updated document bare mapping to clarify that `@WebParam` and `@WebResult` can be used to customize the generated global element names, see section 3.6.2.2.

B.5 Changes Since Early Draft 1

- Added chapter 5 Service APIs.
- Added chapter **??** WSDL 2.0 to Java Mapping.
- Added chapter **??** Java to WSDL 2.0 Mapping.
- Added mapping from Java to `wsdl:service` and `wsdl:port`, see sections 3.8.1, 3.10.1 and 3.11.
- Fixed section 2.4 to allow use of JAXB interface based mapping.
- Added support for document/literal/bare mapping in Java to WSDL mapping, see section 3.6.
- Added conformance requirement to describe the expected behaviour when two or more faults refer to the same global element, see section 2.5.
- Added resolution to issue regarding binding of duplicate headers, see section 2.6.2.1.
- Added use of JAXB ns URI to Java package name mapping, see section 2.1.
- Added use of JAXB package name to ns URI mapping, see section 3.2.
- Introduced new typographic convention to clearly mark non-normative notes.
- Removed references to J2EE and JNDI usage from ServiceFactory description, see section **??**.
- 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 proocol bindings 10.4. The HTTP subsection of this now contains much of the mterial 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 2.3.4.
- Added support for WSDL MIME binding, see section 2.6.3.
- Clarified that fault mapping should only generate a single exception for each equivalent set of faults, see section 2.5.
- Added property for message attachments.
- Removed element references to anonymous type as valid for wrapper style mapping (this doesn’t prevent substitution as orignally thought), see section 2.3.1.2.
- Removed implementation specific methods from generated service interfaces, see section 2.7.
- Clarified behaviour under fault condition for asynchronous operation mapping, see section 2.3.4.5.
- Clarified that additional parts mapped using soapbind:header cannot be mapped to a method return type, see section 2.3.2.
- Added new section to clarify mapping from exception to SOAP fault, see 10.2.2.3.

- Clarified meaning of *other* in the handler processing section, see 9.3.2.
- Added a section to clarify Stub use of RemoteException and JAXRPCException, see 4.2.4.
- 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 **??** and **??**.
- Clarrified SOAP mustUnderstand processing, see section 10.2.1. 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 3.7.

Bibliography

[1] 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.

[2] 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/.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Joshua Bloch. A Metadata Facility for the Java Programming Language. JSR, JCP, August 2003. See http://jcp.org/en/jsr/detail?id=175.

[13] Jim Trezzo. Web Services Metadata for the Java Platform. JSR, JCP, August 2003. See http://jcp.org/en/jsr/detail?id=181.

[14] Jim Knutson and Heather Kreger. Web Services for J2EE. JSR, JCP, September 2002. See http://jcp.org/en/jsr/detail?id=109.

[15] Nataraj Nagaratnam. Web Services Message Security APIs. JSR, JCP, April 2002. See http://jcp.org/en/jsr/detail?id=183.

[16] Farrukh Najmi. Java API for XML Registries 1.0 (JAXR). JSR, JCP, June 2002. See http://www.jcp.org/en/jsr/detail?id=93.

[17] 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.

[18] 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.

[19] 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.

[20] John Cowan and Richard Tobin. XML Information Set. Recommendation, W3C, October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024/.

[21] 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/.

[22] 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/.

[23] 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.

[24] 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/.

[25] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Herve Ruellan. XML-binary Optimized Packaging. Recommendation, W3C, January 2005. http://www.w3.org/TR/xop10/.

[26] 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.

[27] 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.

[28] 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.

[29] Rajiv Mordani. Common Annotations for the Java Platform. JSR, JCP, July 2005. See http://jcp.org/en/jsr/detail?id=250.

[30] Bill Shannon. Java Platform Enterprise Edition 5 Specification. JSR, JCP, August 2005. See http://jcp.org/en/jsr/detail?id=244.

[31] John Barton, Satish Thatte, and Henrik Frystyk Nielsen. SOAP Messages With Attachments. Note, W3C, December 2000. http://www.w3.org/TR/SOAP-attachments.